

软件架构师应该知道的97件事

97 Things Every Software Architect Should Know

Collective Wisdom from the Experts

Richard Monson-Haefel 编

徐定翔 章显洲 译　余晟 审校





O'REILLY®

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

软件架构师应该知道的97件事

这是本与众不同的技术图书。五十多位作者中不乏像尼尔•福特（Neal Ford）、迈克尔•尼加德（Michael Nygard）、比尔•德•霍拉（Bill de hóra）这样杰出的软件架构师，大家分享了多年积累的开发经验和工作准则，内容不限于单纯的技术范畴，还涉及如何与各方沟通、如何降低项目的复杂度、怎样强化开发团队等。有代表性的主题包括：

- 客户需求重于个人简历——尼廷•博万卡（Nitin Borwankar）。
- 关键问题可能不是出在技术上——马克•兰姆（Mark Ramm）。
- 以沟通为中心，坚持简明清晰的表达方式和开明的领导风格——马克•理查兹（Mark Richards）。
- 先确保解决方案简单可用，再考虑通用性和复用性——凯佛林•亨尼（Kevlin Henney）。
- 对最终用户而言，界面就是系统——维纳亚克•赫格德（Vinayak Hegde）。
- 提前关注性能问题——丽贝卡•帕森斯（Rebecca Parsons）。

成为出色的软件架构师要既掌握业务知识又具备技术能力。顶尖的软件架构师看重什么？他们如何完成项目？怎样提高自己的工作水平？推荐您阅读《软件架构师应该知道的97件事》。

理查德•蒙森-哈斐尔是独立软件开发者，曾参加编写《Enterprise JavaBeans》和《Java Message Service》（均由O'Reilly公司出版）。他是企业计算方面的专家，同时擅长设计和开发多点触控应用程序。

图书分类：软件工程

www.oreilly.com

责任编辑：杨绣国

Broadview®
WWW.BROADVIEW.COM.CN

www.phei.com.cn

ISBN 978-7-121-10635-4

9 787121 106354 >

定价：39.80元

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao and Taiwan)

O'REILLY®

O'REILLY®

软件架构师应该知道的 97 件事

97 Things Every Software Architect Should Know

Richard Monson-Haefel 编

徐定翔 章显洲 译

余 晟 审校

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内容简介

优秀的软件架构师应该既掌握业务知识又具备技术能力，做到这一点绝非易事，本书想要探讨的就是这个主题。这是一本真正的开源图书，我们邀请到 50 多位杰出的软件架构师参与写作。大家无偿地分享了各自的工作经验和心得，内容从规避风险的方法到组建团队的技巧，涵盖了架构设计的方方面面。衷心希望这 97 篇文章能激发您的思考，解决您工作中的困惑。

978-0-596-52269-8 97 Things Every Software Architect Should Know © 2009 by O'Reilly Media, Inc. Simplified Chinese edition, jointly published by O'Reilly Media ,Inc. and Publishing House of Electronics Industry, 2009.Authorized translation of the English edition, 2009 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版专有出版权由 O'Reilly Media, Inc. 授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号 图字：01-2009-2863

图书在版编目（CIP）数据

软件架构师应该知道的 97 件事 /（美）蒙森 - 哈斐尔 (Monson-Haefel,R.) 编；徐定翔，章显洲译 . —北京：电子工业出版社，2010.4

书名原文：97 Things Every Software Architect Should Know

ISBN 978-7-121-10635-4

Ⅰ.①软… Ⅱ.①蒙… ②徐… ③章… Ⅲ.①软件设计 Ⅳ.①TP311.5

中国版本图书馆 CIP 数据核字 (2010) 第 056651 号

策划编辑： 徐定翔

责任编辑： 杨绣国

项目管理： 梁　晶

封面设计： Mark Paglietti，张　健

印　　刷： 北京市天竺颖华印刷厂

装　　订： 三河市鑫金马印装有限公司

出版发行： 电子工业出版社

北京市海淀区万寿路173信箱　邮编：100036

开　　本： 720×1000　1/16　**印张：** 14　**字数：** 250千字

印　　次： 2010年4月第1次印刷

定　　价： 39.80元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，

联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在Unix、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是在线出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到 WebSite（第一个桌面 PC 的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

权限声明
Permissions

书中文章的使用许可权也与开放源代码的方式相似。所有文章都遵守知识共享（Creative Commons）协议的第 3 版姓名标示条款（Attribution 3）。读者可以通过本书的主题网站免费阅读所有英文文章，如果大家希望在自己的作品中引用这些文章，必须按照作者或授权人指定的方式保留其姓名标示。此前也有人尝试用开源的方式组织编写图书，但是成书者寥寥无几。我认为主要的原因是缺乏模块化的组织方式。而本书成功付梓恰恰是因为采用了模块化的结构 ——所有文章都能独立成篇，集结成册又是有机的整体。

欢迎与我们联系
How to Contact Us

如果您希望提交关于此书的评论和问题，请按照以下的联系方式与我们联系。

奥莱利技术咨询（北京）有限公司
北京市 西城区 西直门 南大街2号 成铭大厦C座807室
邮政编码：100080
网页：*http://www.oreilly.com.cn*
E-mail：*info@mail.oreilly.com.cn*

O'Reilly Media, Inc
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international/local)
707-829-0104 (fax)

与本书有关的在线信息如下所示：

http://www.oreilly.com/catalog/9780596522698/ （原　书）
http://www.oreilly.com.cn/book.php?bn=9787121106354/ （中文版）

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特1号 湖北信息产业科技大厦1402室

邮政编码：430074

电话： (027)87690813 传真：(027)87690813转817

读者服务网页：*http://bv.csdn.net*

E-mail：

reader@broadview.com.cn（读者信箱）

bvtougao@gmail.com（投稿信箱）

致谢

Acknowledgments

这本书并非凭空而成，我希望按时间顺序感谢那些在策划和组稿阶段给予我帮助的朋友。首先是 Jay Zimmerman，是他建议我在 No Fluff, Just Stuff 的研讨会上发表题为“软件架构师应该知道的 10 件事”的演讲；Bruce Eckel，感谢他维护的邮件列表，这本书的雏形是在那里讨论出来的。Jeremy Meyer 建议我扩展演讲内容，编写成书；Nitin Borwankar 提出以 Wiki 为写作平台，对所有人开放，还有 Bruce 邮件列表中的朋友，他们无偿地奉献了自己的时间和最初的文章。我还要感谢十几位文章未能入选的软件架构师，感谢他们认真地参与了这个创作过程。这些文章真是难以取舍，在此要向大家致以深深的敬意。

我还要感谢 O'Reilly 公司支持我的策划想法，允许我们尝试新的写作方式。O'Reilly 公司同意将所有内容开源(遵守知识共享协议的第 3 版姓名标示条款)，公开放到在网站上。我要感谢几位 O'Reilly 的员工：Mike Loukides、Mike Hendrickson、Laura Painter、Laurel Ackerman，没有他们的帮助，这本书不可能面市。

O'Reilly 和我正在组织若干以“97 件事”为题的图书。我们希望出版一个新颖的图书系列，充分挖掘各领域专家的集体智慧。目前还有三本书正在制作，分别是项目管理、软件开发、数据架构。

希望这本书能激发大家的思考，同时期待大家在后续的“97 件事”系列图书中分享思考的硕果。

祝大家诸事如意！

——理查德·蒙森-哈斐尔（Richard Monson-Haefel）

“97 件事”系列图书的编辑

译者序

97 个水晶切面，折射软件架构师的实践智慧

近十数年，全球软件产业已经取得了显著的进步，软件从业人员的数量不断增加，软件项目的规模和复杂度不断攀升，软件开发组织的人员结构也因专业不断细分而日趋复杂。

因兴起时间较晚，发展的时间相对还很短，软件业从建筑业和制造业等其他成熟的工程学科中借用了许多概念和隐喻。“软件架构”和“软件架构师”，便是借用了建筑设计中的概念。从 UML（统一建模语言，Unified Modeling Language）和软件模式（Pattern）相关的早期著作中，可以清晰地看到这种概念移用。

借用其他行业的概念会带来一些消极影响。比如在早期阶段，一些“软件工程”研究人员和软件项目管理人员，试图以建筑业的项目管理视角和技术来管理软件项目，制造出了“瀑布式”软件开发过程，给软件业带来了长期的不良影响。但是，经过实践验证，“软件架构”确实推动了软件开发技术的发展，为业界广为采纳并日益受到重视。

现如今，“软件架构师”已成为许多软件开发组织职位模型（Job Model）中的标准设置。“软件架构师”也已成为众多软件开发人员梦寐以求的职业巅峰目标之一。

业界许多大师和专家总结分享了他们在软件架构设计方面的技术和经验。在软件技术书籍市场上，已经可以找到许多与“软件架构”主题相关的书籍。

不过，现有的软件架构书籍从技术视角进行总结阐述的居多，如已有 5 卷本行世的《基于模式的软件架构（Pattern-Oriented Software Architecture，POSA）》，主要阐述的是软件架构的风格、解决特定问题或特定领域的软件架构设计技术与模式等。这些书籍当然都非常有益，甚至，可以说成为了软件架构师的必读书目。

但是，即使已经将这些软件架构设计的技术、模式烂熟于胸，可能还无法保证你能够成为优秀的软件架构师。

为何会这样?

正如《建筑十书》的作者、古罗马著名建筑师维特鲁威所说："理想的建筑师应该既是文学家又是数学家，他还应通晓历史，热衷于哲学研究，精通音乐，懂得医药知识，具有法学造诣，深谙天文学及天文计算。"

优秀的软件架构师，必须多才多艺、成熟练达、经验丰富，具备极强的洞察力，能够领导和提升软件开发团队，去构建整齐有序、均衡合理、可持续发展演化的虚拟数字世界中的伟大建筑——优秀杰出的软件产品。

本书提供了分享软件架构知识的新方式，拓宽了阐述软件架构艺术的视角，总结了 50 多位经验丰富的软件架构师的实践经验，范围覆盖了软件架构师的职业操守、技术技能、思维模式、领导力、和客户的沟通交流、权衡利弊的平衡感等主题。

全书由 97 篇格言式散文构成，没有高调的说教，没有抽象的术语，而是以平实、幽默、智慧的笔触，将他们认为对成为优秀软件架构师而言至为重要的精髓和盘托出。全书犹如一块玲珑剔透的水晶，97 个切面折射出来的都是出自一线软件架构师的专业智慧。

本书第 1~49 篇由徐定翔翻译，第 50~97 篇由章显洲翻译。我们在翻译的过程中得到了许多人的帮助。余晟认真审校了全书的译稿，细致地指出了我们翻译不当之处；郑兆昭老师对书稿的润色让我们受益匪浅；统稿编辑白爱萍一次又一次容忍了我们修改定稿的要求。最后要感谢家人给予我们的宽容和支持，使我们能够安心埋首于书稿中。

由于时间仓促，加上我们水平有限，书中难免有翻译错漏之处，有些词语的翻译也许有争议，请广大读者和同行不吝指正。大家可以通过邮箱 *97things4architect@gmail.com* 和我们联系。

译者

2010 年春

目录

Contents

前言 I

客户需求重于个人简历 2

尼廷·博万卡（Nitin Borwankar）

简化根本复杂性，消除偶发复杂性 4

尼尔·福特（Neal Ford）

关键问题可能不是出在技术上 6

马克·兰姆（Mark Ramm）

以沟通为中心，坚持简明清晰的表达方式和开明的领导风格 8

马克·理查兹（Mark Richards）

架构决定性能 10

兰迪·斯塔福德（Randy Stafford）

分析客户需求背后的意义 12

埃纳尔·兰德雷（Einar Landre）

起立发言 14

乌迪·大汉（Udi Dahan）

故障终究会发生 16
迈克尔·尼加德（Michael Nygard）

我们常常忽略了自己在谈判 18
迈克尔·尼加德（Michael Nygard）

量化需求 20
基思·布雷思韦特（Keith Braithwaite）

一行代码比五百行架构说明更有价值 22
艾利森·兰德尔（Allison Randal）

不存在放之四海皆准的解决方案 24
兰迪·斯塔福德（Randy Stafford）

提前关注性能问题 26
丽贝卡·帕森斯（Rebecca Parsons）

架构设计要平衡兼顾多方需求 28
兰迪·斯塔福德（Randy Stafford）

草率提交任务是不负责任的行为 30
尼克拉斯·尼尔森（Niclas Nilsson）

不要在一棵树上吊死 32
基思·布雷思韦特（Keith Braithwaite）

业务目标至上 34
戴夫·缪尔黑德（Dave Muirhead）

先确保解决方案简单可用，再考虑通用性和复用性 36
凯佛林·亨尼（Kevlin Henney）

架构师应该亲力亲为 38
约翰·戴维斯（John Davies）

持续集成 40
大卫·巴特利（David Bartlett）

避免进度调整失误 42
诺曼·卡诺瓦利（Norman Carnovale）

取舍的艺术 44
马克·理查兹（Mark Richards）

打造数据库堡垒 46
丹·恰克（Dan Chak）

重视不确定性 48
凯佛林·亨尼（Kevlin Henney）

不要轻易放过不起眼的问题 50
戴夫·奎克（Dave Quick）

让大家学会复用 52
杰里米·迈耶（Jeremy Meyer）

架构里没有大写的“I” 54
戴夫·奎克（Dave Quick）

使用“一千英尺高”的视图 56
埃里克·多伦伯格（Erik Doernenburg）

先尝试后决策 58
埃里克·多伦伯格（Erik Doernenburg）

掌握业务领域知识 60
马克·理查兹（Mark Richards）

程序设计是一种设计.....62
埃纳尔·兰德雷（Einar Landre）

让开发人员自己做主.....64
菲利普·尼尔森（Philip Nelson）

时间改变一切.....66
菲利普·尼尔森（Philip Nelson）

设立软件架构专业为时尚早.....68
巴里·霍金斯（Barry Hawkins）

控制项目规模.....70
大卫·奎克（Dave Quick）

架构师不是演员，是管家.....72
巴里·霍金斯（Barry Hawkins）

软件架构的道德责任.....74
迈克尔·尼加德（Michael Nygard）

摩天大厦不可伸缩.....76
迈克尔·尼加德（Michael Nygard）

混合开发的时代已经来临.....78
爱德华·加森（Edward Garson）

性能至上.....80
克雷格·罗素（Craig Russell）

留意架构图里的空白区域.....82
迈克尔·尼加德（Michael Nygard）

学习软件专业的行话.....84
马克·理查兹（Mark Richards）

具体情境决定一切 86
爱德华·加森（Edward Garson）

侏儒、精灵、巫师和国王 88
埃文·考夫斯基（Evan Cofsky）

向建筑师学习 90
基思·布雷思韦特（Keith Braithwaite）

避免重复 92
尼克拉斯·尼尔森（Niclas Nilsson）

欢迎来到现实世界 94
格雷戈尔·侯珀（Gregor Hohpe）

仔细观察，别试图控制一切 96
格雷戈尔·侯珀（Gregor Hohpe）

架构师好比两面神 98
大卫·巴特利（David Bartlett）

架构师当聚焦于边界和接口 100
埃纳尔·兰德雷（Einar Landre）

助力开发团队 102
蒂莫西·海伊（Timothy High）

记录决策理由 104
蒂莫西·海伊（Timothy High）

挑战假设，尤其是你自己的 106
蒂莫西·海伊（Timothy High）

分享知识和经验 108
保罗·W·霍默（Paul W. Homer）

模式病.....110
查德·拉·瓦因（Chad La Vigne）

不要滥用架构隐喻.....112
戴维·英格（David Ing）

关注应用程序的支持和维护.....114
门西蒂西·卡斯珀（Mncedisi Kasper）

有舍才有得.....116
比尔·德·霍拉（Bill de hÓra）

先考虑原则、公理和类比，再考虑个人意见和口味.....118
迈克尔·哈默（Michael Harmer）

从“可行走骨架”开始开发应用.....120
克林特·尚克（Clint Shank）

数据是核心.....122
保罗·W·霍默（Paul W. Homer）

确保简单问题有简单的解.....124
查德·拉·瓦因（Chad La Vigne）

架构师首先是开发人员.....126
迈克·布朗（Mike Brown）

根据投资回报率（ROI）进行决策.....128
乔治·马拉米迪斯（George Malamidis）

一切软件系统都是遗留系统.....130
戴夫·安德森（Dave Anderson）

起码要有两个可选的解决方案.....132
蒂莫西·海伊（Timothy High）

理解变化的影响 134
道格·克劳福德（Doug Crawford）

你不能不了解硬件 136
卡迈尔·威克拉玛纳亚克（Kamal Wickramanayake）

现在走捷径，将来付利息 138
斯科特·麦克菲（Scot Mcphee）

不要追求“完美”，“足够好”就行 140
格雷格·纽伯格（Greg Nyberg）

小心“好主意” 142
格雷格·纽伯格（Greg Nyberg）

内容为王 144
朱宾·沃迪亚（Zubin Wadia）

对商业方，架构师要避免愤世嫉俗 146
查德·拉·瓦因（Chad La Vigne）

拉伸关键维度，发现设计中的不足 148
斯蒂芬·琼斯（Stephen Jones）

架构师要以自己的编程能力为依托 150
迈克·布朗（Mike Brown）

命名要恰如其分 152
萨姆·加德纳（Sam Gardiner）

稳定的问题才能产生高质量的解决方案 154
萨姆·加德纳（Sam Gardiner）

天道酬勤 156
布赖恩·哈特（Brian Hart）

对决策负责 158
周异（Yi Zhou）

弃聪明，求质朴 160
埃本·休伊特（Eben Hewitt）

精心选择有效技术，绝不轻易抛弃 162
查德·拉·瓦因（Chad La Vigne）

客户的客户才是你的客户！ 164
埃本·休伊特（Eben Hewitt）

事物发展总会出人意料 166
彼得·吉拉德莫斯（Peter Gillard-Moss）

选择彼此间可协调工作的框架 168
埃里克·霍索恩（Eric Hawthorne）

着重强调项目的商业价值 170
周异（Yi Zhou）

不仅仅只控制代码，也要控制数据 172
查德·拉·瓦因（Chad La Vigne）

偿还技术债务 174
伯克哈特·赫夫纳盖尔（Burkhardt Hufnagel）

不要急于求解 176
埃本·休伊特（Eben Hewitt）

打造上手（Zuhanden）的系统 178
基思·布雷思韦特（Keith Braithwaite）

找到并留住富有激情的问题解决者 180
查德·拉·瓦因（Chad La Vigne）

软件并非真实的存在.....182
查德·拉·瓦因（Chad La Vigne）

学习新语言.....184
伯克哈特·赫夫纳盖尔（Burkhardt Hufnagel）

没有永不过时的解决方案.....186
理查德·蒙森-哈费尔（Richard Monson-Haefel）

用户接受度问题.....188
诺曼·卡诺瓦利（Norman Carnovale）

清汤的重要启示.....190
埃本·休伊特（Eben Hewitt）

对最终用户而言，界面就是系统.....192
维纳亚克·赫格德（Vinayak Hegde）

优秀软件不是构建出来的，而是培育起来的.....194
比尔·德·霍拉（Bill de hÓra）

索引.....196

前言

Preface

软件架构师是 IT 行业里独一无二的职业，既要精通开发技术和软件平台，又要熟悉客户的业务。优秀的软件架构师应该同时掌握业务知识和技术能力，做到这一点绝非易事，本书要探讨的就是这个主题。

在这本书中，来自世界各地的软件架构师分享了各自的工作经验和心得，内容从规避风险的方法到组建团队的技巧，涵盖了架构设计的方方面面。这些作者都是业界功成名就的架构师，他们分享的经验既可供同行参考，也适合新手阅读。读者可以像吃自助餐一样，从中自由挑选感兴趣的主题。

我衷心希望这本书能激发读者的思考，成为帮助大家的向导。从事软件架构设计可能是 IT 行业里难度最大的工作，期待大家借助本书和本书的主题网站进一步分享对这份工作的见解和领悟。

这本书没有采用传统的方式组织内容。五十多位软件架构师参与了本书的写作，他们把工作中总结出来的观点和建议无偿地献给了这本书。这是一本真正的开源图书，每位作者独立写作投稿，然后编辑在作者的协助下审阅、编辑加工稿件，最后从所有的稿件中挑选出最优秀的文章出版。整个过程与开源软件项目的组织方式很相像，只不过软件项目贡献的是代码，而这本书贡献的是知识和智慧。

客户需求重于个人简历

Don't Put Your Resume Ahead of the Requirements

尼廷·博万卡（Nitin Borwankar）



作为工程师，我们常常要向客户推荐技术、手段，甚至方法论来解决问题。但有时我们心里不是想寻求解决问题的最佳方案，而是希望借此丰富自己的简历。这样做很可能得不偿失。

积累一批满意的客户，选择切合实际的技术解决他们的难题，让他们乐于推荐你，才是最好的履历。信誉远胜过时髦的编程技巧和流行的范式。掌握最新的技术趋势，与时俱进固然重要，但不能让客户为此买单。作为架构师，职业操守绝不能忘。公司托付重任给你，是期望你恪尽职守，不受利益诱惑。如果你觉得项目不够尖端，挑战性不足，无法满足职业发展的需要，大可另栖高枝，另谋高就。

万一你别无选择，必须参与这样的项目，不要为简历所累。忍痛割爱放弃时髦光鲜的方案确实不容易（哪怕它们并不适合当前的项目），但只有脚踏实地替客户着想，最后才能皆大欢喜。

选择正确的解决方案可以降低项目的压力，团队工作起来更开心，客户也更满意。你会有更充裕的时间，既可以钻研现有技术，也可以利用空闲时间学习新知识，甚至重拾向往已久的业余爱好。家人察觉你的变化后，也会感到欣慰。

把客户的长远需求摆在自己的短期利益之上，才能立于不败之地。

作者简介：

尼廷·博万卡 20 世纪 90 年代初曾先后就职于 Ingres 公司和 Sybase 公司。他曾参与开发最早的网络数据库应用，那时的开发语言是 SybPerl 和 OraPerl，之后他又投身研究 Enterprise Java。他曾积极参与 New-EDI 项目——一个基于 IETF 标准的因特网电子数据交换系统。1994 年以来，他一直作为独立顾问和研究者关注企业数据集成和消息传递的研究。他目前的兴趣包括研究数据库模式（schemas）用来实现企业应用中的大众分类方法（folksonomy），以及将社交网络运用于企业应用时的底层数据库问题。他是数据可移植性（Data Portability）策略小组的成员，负责起草有关用户数据权利的最终用户许可协议。他撰写了多篇数据库方面的文章，发表在 GigaOm.com 和他的博客（*http://tagschema.com*）上。他还拥有一项跨越可信赖边界传递协同消息的专利。

简化根本复杂性，消除偶发复杂性

Simplify Essential complexity; Diminish Accidental Complexity

尼尔·福特（Neal Ford）



根本复杂性（essential complexity）指的是问题与生俱来的、无法避免的困难。比如，协调全国的空中交通就是一个“天生的”复杂问题，必须实时跟踪每架飞机的位置（包括飞行高度）、航速、航向和目的地，才能预防空中和地面上的冲突。像天气骤变这样的情况会令航班计划全盘失效，航班时刻表必须适应不断变化的环境才能避免乘客滞留。

与之相反，偶发复杂性（accidental complexity）是人们解决根本复杂性的过程中衍生的。目前陈旧的空中交管系统，就是一个偶发复杂性的例子。系统设计的初衷是管理数以千计的飞机参与的交通活动，即解决根本复杂性，但是解决方案本身带来了新的问题。事实上，目前正在服役的空中交管系统，其复杂臃肿已经到了难以改善的地步。在全球多数地区，空中交管系统仍然在使用三十多年前的技术。

许多软件框架和厂商提供的“解决方案”都表现出偶发复杂性的症状。解决特定问题的框架很管用，但设计过度的框架增加的复杂性反而超过了它应该缓解的复杂性。

开发人员痴迷于复杂的问题，好比飞蛾喜欢扑火。谁能拒绝迅速解决复杂问题带来的快感？但是开发人员应该解决问题，而不是解谜取乐。在大型软件项目中，关注根本复杂性，消除偶发复杂性，抽丝剥茧制订解决方案，才是真正的挑战。

该怎么做呢？应该尽量选择源自实际项目的框架，警惕那些象牙塔里的产品；分析方案中有多少代码直接用来解决业务问题，有多少只是用来实现用户与应用之间的交互；谨慎使用软件厂商在幕后推动的方案，它们并非一无是处，但往往包含偶发复杂性；要量体裁衣，为问题制订“合身”的解决方案。

架构师的责任在于解决问题的根本复杂性，同时避免引入偶发复杂性。

作者简介：

尼尔·福特是 ThoughtWorks 公司的软件架构师和文化基因探索者（meme wrangler）。ThoughtWorks 是一家全球性的 IT 咨询公司，专注于端到端的软件开发与交付。除了开发软件，尼尔还编写教材，撰写杂志文章，制作课件和视频 DVD，并出版了五本著作。他经常参加各种会议并发表演讲。更多详情，请访问他的个人网站（*http://www.nealford.com*）。

关键问题可能不是出在技术上

Chances Are, Your Biggest Problem Isn't Technical

马克·兰姆（Mark Ramm）



简单的项目（比如工资管理系统）也会翻船，而且这不是个别情况。

为什么？难道是因为我们用错了技术吗？因为错选了 Ruby 而不是 Java，错选了 Python 而不是 Smalltalk？或者选择了 Postgres 而不是 Oracle？还是本该用 Linux 时，错选了 Windows？一旦项目失败，技术往往沦为替罪羊。但是有多少问题真的是 Java 无法胜任的呢，这种可能性有多大？

大多数项目是由人完成的，人才是项目成败与否的基础。如何帮助团队成员完成项目，这个问题很值得静下心来好好思考。

如果团队里有人工作方式不正确，拖项目的后腿怎么办？有一种非常古老但很完善的技术可以帮助你解决问题。它可能是人类历史上最重要的技术创新，这就是对话。

仅仅了解对话的用途还不够。学会尊重他人，给予团队成员充分的信任，是聪明的架构师获得成功必须掌握的核心技能。

关于对话的技巧非常多，但有几个简单的技巧可以显著改善对话的效果：

- 不要把对话当成对抗。

 如果你能看到他人的优点，并把沟通视为请教问题的机会，就会有所收获，同时也能避免引起对方的戒备之心。

- 不要带着情绪与人沟通。

 当你处于愤怒、沮丧、烦恼，或者慌张的情绪中时，对方很可能会误认为你的举动不怀好意。

- 尝试通过沟通设定共同的目标。

 有些人开会时喋喋不休影响别人发言，与其命令他闭嘴，不如请他协助你提高其他人的参与度。告诉他有些同事比较内向，发言前需要安静地理清思路。请他在每次发言之前稍做等待，让同事有机会表达意见。

首先与同事达成一致的目标，把处理冲突和矛盾的过程视为学习的机会，控制住自己的情绪，那么每次沟通都会有所收获，你会做得越来越好。

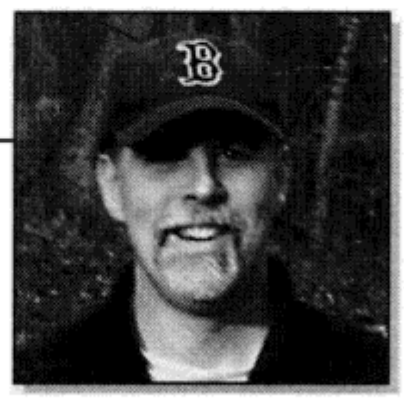
作者简介：

马克·兰姆是 TurboGears 2 开源项目领头人。这位老兄热爱 Python，喜欢挑战，他干过各种奇怪的工作，除了软件架构师和网络管理员，他还捕过龙虾，在飞车党酒吧当过清洁工。他目前致力于开发软件工具，帮助专业程序员和业余程序员更好地开展工作。

以沟通为中心，坚持简明清晰的表达方式和开明的领导风格

Communication Is King; Clarity and Leadership, Its Humble Servants

马克·理查兹（Mark Richards）



软件架构师普遍喜欢坐在象牙塔里，命令开发人员执行他的命令和技术决策。这很容易引发大家的抵触情绪，造成团队不和，甚至导致产品与最初的需求相去甚远。软件架构师应该想办法提高自己的沟通技巧，帮助大家理解项目的目标。关键在于明确有效的沟通和开明的领导风格。

沟通必须简明清晰。没有人愿意阅读冗长的架构决策文档，架构师言简意赅地表达观点是项目成功的必要条件。项目启动之初，凡事能简则简，千万不要一头扎入冗长的 Word 文档里。可以借助工具，比如简单的 Visio 图表来表达你的想法，尽量画简单些，毕竟时过境迁，想法总会变化。非正式的白板会议是另一种有效的沟通手段，把开发人员（还有其他架构师）召集起来，在白板上写下你的想法，比任何方法都来得有效。此外，别忘了随身携带相机，拍下白板上的内容，通过 Wiki 在团队内共享，毕竟会后回忆讨论的内容并不容易。扔掉冗长的 Word 文档，想办法让大家接受你的观点，最后别忘了详细记录讨论结果。

还有，架构师往往忽略了自己也是领导者。作为领导者，我们必须获得同伴的尊敬才能顺利开展工作。如果开发人员对项目蓝图和决策过程一无所知，必定会产生隐患。安排一位你信得过的开发人员牵头，创造良好的合作环境，请大家共同验证你的架构决策。让开发人员参与架构的制订过程，他们才会买你的账。与其和开发人员对着干，不妨与他们合作。请记住，所有的团队成员（包括质量控制小组、业务分析员、项目经理，以及开发人员）都渴望明确的沟通和开明的领导。只有这样才能改善沟通效果，建立团结健康的工作环境。

以沟通为中心，坚持简明清晰的表达方式和开明的领导风格。

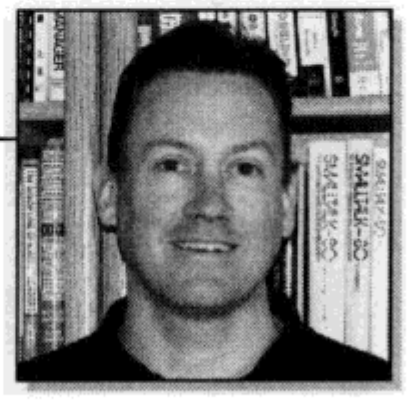
作者简介：

马克·理查兹是 Collaborative Consulting 有限责任公司的主管和高级解决方案架构师，他的主要工作是利用 J2EE 和相关技术为金融服务行业设计并提供大规模面向服务的架构。自从 1984 年进入软件业以来，他在 J2EE 的架构和开发，面向对象设计和开发，以及系统集成方面积累了丰富的经验。

架构决定性能

Application Architecture Determines Application Performance

兰迪·斯塔福德（Randy Stafford）



架构决定应用的性能，似乎是大家都明白的道理，但是事实并非如此。有些架构师认为简单地更换底层软件架构（Software Infrastructure）就足以解决应用的性能问题。他们很可能轻信了“经测试产品性能超出竞争对手 25%”一类的商业噱头。假设某产品完成特定操作耗时 3 毫秒，竞争对手需要 4 毫秒，这 1 毫秒（25%）的优势如果放到一个性能效率极低的架构里，几乎可以忽略不计。架构是决定应用性能的根本因素。

撇开 IT 经理和厂商的测试团队，另一些人（比如产品技术支持部门和应用性能管理文献的作者）则建议直接通过“调优”（Tuning）架构来解决问题，例如改变内存的分配方法、调整连接池或线程池的大小，等等。但是，如果应用的部署方案满足不了预期的负载（load）要求，或者应用软件的功能架构不能充分利用计算资源，那么无论怎样“调优”都无法带来理想的性能和可伸缩（scalability）特性。这时必须重新设计架构的内在逻辑和部署策略。

归根结底，所有产品和架构都必须遵循分布式计算和物理学的基本原理：运行应用和产品的计算机性能有限，通过物理连接和逻辑协议实现的通信必然有延时。因此，应该承认架构才是影响应用性能和可伸缩性的决定因素。性能参数是无法简单地通过更换软件，或者“调优”底层软件架构来改善的，我们必须在架构的设计（或重新设计）上投入更多精力。

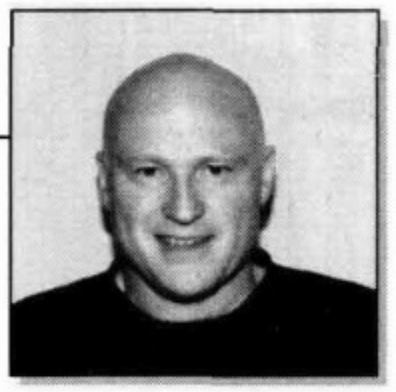
作者简介：

兰迪·斯塔福德拥有 20 年从事软件行业的经验，他身兼多职，既是开发者、分析师、架构师，又是经理、顾问、作家/投稿人。目前他是甲骨文公司中间件精英团队的成员，负责帮助全球范围内的客户和组织验证概念项目，审查架构，解决生产问题，他致力于研究网格、SOA、性能优化、高可用性，以及 JEE/ORM。

分析客户需求背后的意义

Seek the Value in Requested Capabilities

埃纳尔·兰德雷（Einar Landre）



顾客和最终用户通常提出的所谓需求，只是他们心目中可行的解决方案，并不是问题唯一的解决途径。F-16“战隼”（战斗机）的设计师哈里·希拉克尔（Harry Hillaker）曾就此给出过一个经典的案例。军方最初对 F-16 的设计需求是：飞行速度在 2~2.5 马赫（译注 1）之间的低成本轻型战斗机。要知道当飞行速度由 1 倍音速变为 2 倍音速时，空气阻力会增至原来的 4 倍，考虑到因此所需要的动力，及其对机身重量的苛刻条件，满足这样的需求，即使是在今天也绝非易事。

设计团队追问军方为什么需要 2~2.5 马赫的速度。答复是“为了迅速撤离战场”。了解真正的需求后，设计团队对症下药提出了有效的解决方案：通过提升推力重量比（Thrust-to-weight Ratio），改善战斗机的加速性能和机动性能。用灵巧性取代了最初对速度的需求。

软件开发也应该借鉴这条经验。架构师可以通过询问客户，分析客户要求的功能和需求的真正意义，定位真正的问题，从而提出比客户的建议更好、成本更低的解决方案。通过关注问题的真正含义，理顺需求的轻重缓急：把最有价值的需求摆在第一位。

译注 1：马赫（Mach）是速度单位，1 马赫约等于 340 米/秒，即音速。

该怎么做呢？敏捷宣言提供了答案：“客户合作重于合同谈判（Collaboration over contract）”。具体来说，架构师应该通过与客户面对面的交流，关注客户的需求，引导客户回答“为什么”的问题。要知道说明“为什么”并不容易，因为客户往往觉得问题是不言而喻的。此外，避免与客户讨论技术上的具体实现，这样做会喧宾夺主，因为此时应该关注的是客户的问题，而不是软件开发的问题。

作者简介：

埃纳尔·兰德雷（Einar Landre）25 年来一直从事着与软件相关的工作，他当过程序员、架构师、经理人和顾问，还写书和主持会议。目前，埃纳尔正忙于 StatoilHydro 公司的企业应用服务项目，他参与开发关键业务应用，评审架构，以及改进开发过程。他擅长面向服务的架构（SOA），领域驱动设计（domain-driven design）和利用多代理结构设计大规模软件密集型的联网系统。

起立发言

Stand up!

乌迪·大汉（Udi Dahan）



许多架构师都是从技术岗位上成长起来的，他们擅长与机器打交道。然而架构师更需要与人打交道，无论是劝说开发人员接受具体的设计模式，还是向管理层解释购买中间件的利弊，沟通都是达成目标的核心技能。

虽然架构师对项目的影响很难界定，但有一点是清楚的：无论架构师的建议多么正确，如果开发人员和管理层都不买账，架构师就无法取得事业上的成功。有经验的架构师都很重视“推销”自己的想法，也明白有效沟通的重要性。

介绍人际沟通诀窍的书不少，我只想向大家介绍一个简单实用的技巧，它可以显著地改善沟通效果，让大家的工作更上一层楼。在两人以上的场合发表意见时，请站起来。无论是正式的设计审查，还是借助图表进行非正式的讨论，起立发言非常重要，尤其是当其他人坐着的时候。

当你站立时，无形中增添了一种权威和自信，自然就控制了场面。听众不会轻易打断你的发言。这些都会让你的发言效果大为改观。

你会发现，站立时可以更好地利用双手和肢体语言。在十人以上的场合，起立发言方便你与每位听众保持视线接触。眼神交流、肢体语言等表达方式在沟通中的作用不可小觑。起立发言还可以让你更好地控制语气、语调、语速和嗓门，让你的声音传得更远。当你讲到重点内容时，注意放慢语速。发声技巧也能显著改善沟通效果。

让沟通事半功倍，起立发言是最简单、有效的方法！

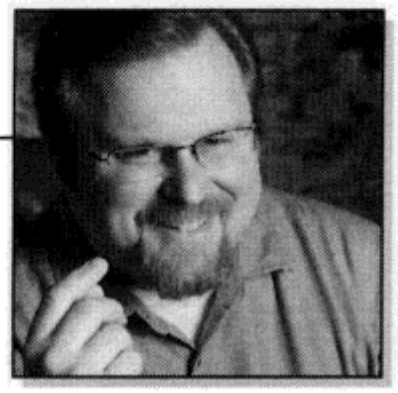
作者简介：

乌迪·大汉是位“软件精简主义者（The Software Simplist）”。他凭借在解决方案架构（Solutions Architecture）方面的造诣，连续三年被微软授予令人羡慕的 MVP 称号。乌迪是微软的 WCF、WF 和 Oslo 技术顾问，他也是微软软件工厂计划咨询委员会的成员，参加了微软模式与实践（Patterns & Practices）组的 Prism 项目。他在全球各地提供培训、指导和高端架构咨询服务，尤其擅长设计面向服务的、可伸缩的、安全的.NET 架构。

故障终究会发生

Everything Will Ultimately Fail

迈克尔·尼加德（Michael Nygard）



硬件会出错，于是我们增加冗余资源来提升系统的可靠性。这样做虽然可以避免由于单点故障引起的系统错误，但同时也增加了至少有一台设备出错的概率。

软件会出错，由软件构成的应用程序自然也会出错，于是我们增加额外的监控程序，好在应用失效时报警。但是监控程序也是软件，一样会出错。

人无完人，我们也会犯错，所以我们把操作、诊断和处理都变成自动化。可是自动化虽然降低了主动犯错的概率，却增加了错误被忽略的概率。何况任何自动化系统应付环境变化的能力都比不上人类。

于是我们又为自动化增加监控，结果是更多的软件，导致更高的故障率。

计算机网络由硬件、软件和长距离的线路构成，当然也会出错。实际上，即便网络工作正常，由于状态空间的无限性，网络的行为也是不可预测的。独立部件的行为可以确定，但是所有部件组合起来，就会无法避免地出现混沌现象。

所有用来避免故障的安全机制都会带来新形式的故障。集群软件可以把应用程序从故障服务器转移到正常的服务器，但如果集群网络功能失常，又会出现“网络分区综合症（split-brain syndrome）”（译注 1）。

译注 1：split-brain syndrome 原意是裂脑综合症，指两个脑半球的感觉及运动功能的连接被切断，以致患者丧失日常生活自理能力的病症。这里用来比喻网络分区之间正常通信失效的情况。

别忘了三哩岛核电站泄漏事故（Three Mile Island accident）（译注 2）是由于减压阀引起的，而减压阀本身是用来避免过压故障的安全机制。

既然系统必然会出错，我们该怎么办呢？

应当承认系统中必然存在着不同形式的故障隐患，无论如何都无法彻底消灭。如果你否认这个事实，管理故障和限制故障就无从谈起。只有承认这一点，才能针对特定的故障设计对策，正如汽车工程师知道交通事故无法避免，所以设计撞击缓冲区（crumple zones）来保护乘客一样，你也可以设计预防措施来限制故障，保护系统其余部分。

如果不事先设计好防范故障的模型，就无法应对威胁系统安全的意外情况。

作者简介：

迈克尔·尼加德著有《Release It! Design and Deploy Production-Ready Software》（Pragmatic Bookshelf 出版社），该书荣获 2008 年 Jolt 生产力大奖（Jolt Productivity award）。他的博客（*http://www.michaelnygard.com/blog*）上有更多文章可供阅读。

译注 2：Three Mile Island accident 指 1979 年发生在美国宾夕法尼亚州三哩岛的核电站泄漏事故。事故最初由二号反应堆的辅助回路冷凝水泵故障引起，导致二号反应堆内温度和压力上升，触发堆内的减压阀开启。卸压后，由于减压阀故障，阀门未能按预期自动关闭，进一步导致冷却水大量溢出，堆心温度上升。待工作人员发现问题所在的时候，47%的堆心燃料已经融毁并发生泄漏，幸好有防护外壳阻挡，未造成人员死亡，但经济损失超过 10 亿美元。

我们常常忽略了自己在谈判

You're Negotiating More Often Than You Think

迈克尔·尼加德（Michael Nygard）



我们都面临过削减预算的要求。如果资金运转捉襟见肘，技术方案只能委曲求全。比如下面的情景：

“我们真的需要这东西吗？”项目投资人发难道。

要知道“这东西”很可能是系统运行必不可少的条件，比如：软件许可证、冗余服务器、离站备份系统，甚至供电设备。更可气的是，投资人总是一副教训的口吻，仿佛只有他懂得钱要花在刀刃上，我们都是毛头小子，就会浪费钱买漫画书和泡泡糖。

答案明摆着是“真的需要”，但很少有人回答得这么干脆。

毕竟我们都是工程师出身，搞工程就是要统筹全局、权衡利弊、适当妥协。例如我们知道只要为数据中心配备发电机，再招几个廉价的实习生，就不必购买奢侈的供电设备。所以我们不会干脆地说“真的需要”，多半会这样回答：“好吧，不买第二台服务器也行，只不过在对系统进行例行维护时必须停机。另外，奇偶校验位翻转也会导致系统崩溃，不过这个问题可以通过购买带奇偶校验的内存来解决。剩下来我们就只用对付操作系统的崩溃了，操作系统大约 3.9 天崩溃一次，解决办法是每晚重启一次系统。当然，这项工作可以找实习生来做。”

虽然句句属实，但绝对是对牛弹琴。投资人想听的只是“好吧”两个字，对后面的话毫无兴趣。

问题出在你没有认清自己的角色，你还是把自己当成工程师，而项目投资人明白他在跟你谈判。工程师总是想尽办法寻求合作，谈判者则绞尽脑汁占得先机。谈判时，绝不能在对方的第一个要求上妥协让步。其实，我们应该这样回答“真的需要吗”这类问题：

“单台服务器每天至少会崩溃三次，系统负载加重的话情况会更严重，没有第二台服务器，我们无法保证你给董事会做演示时一切正常。说实话，我们至少需要四台服务器，两个一组构成两组高可用服务器组，这样可以在需要时断开其中一组，而不必被迫关闭系统。即使剩下的一组中又有一台服务器意外崩溃，也不会影响系统的正常运行。”

当然你并不需要第三台或第四台服务器，这只是谈判的策略，好把对方的话题引开，强调你已经勉力而为，如履薄冰，然后提出更多要求。顺便提一下，如果你真的得到了两台额外的服务器，一台可以用来做产品的质量保证（QA），另一台可以留着随时备用。

作者简介见第 17 页。.

量化需求

Quantify

基思·布雷思韦特（Keith Braithwaite）



“速度快”不能算作需求，“响应灵敏”和“可扩展”也不能算需求，因为我们无法客观地判断是否满足了这样的条件。但这些又确实是用户想要的。架构师的工作在很大程度上是要平衡这些需求之间的不可避免的冲突和矛盾，同时又使系统尽可能地满足它们。如果缺乏客观的标准，架构师就只能任凭挑剔的用户和偏执的程序员摆布（“还不够快，我拒绝接受”和“还不够快，我不能发布”是他们的口头禅）。

在记录需求的过程中，经常会出现模糊的描述，比如“灵活”、“可维护”等。实践证明，所有这些描述都可以量化，并设定相应的检测标准（甚至连“易于使用”这样的需求也可以量化，只是要费些工夫）。没有量化的需求会导致用户验收系统时缺乏依据，架构师不知所措，开发工作失去正确的指导。

我们可以通过一些简单的问题来量化需求，例如：数量有多少？在什么阶段？有多频繁？不能超过多长时间？增加还是减少？占多大比例？如果得不到答案，需求就不明确。这些答案应该包含在系统的业务策划方案里，否则还得费不少脑筋。如果架构师无法从业务部门那里得到这些数字，应该先反思原因，然后再想办法。下次再有人对系统提出“可伸缩（scalable）”的要求，一定要问清楚新用户从何而来，为什么数量会增加，以及何时增加，会增加多少。拒绝“很多”、“马上”这类模棱两可的答案。

对那些无法明确量化的指标也必须给出一个描述范围，比如：上限、下限等。没有范围，就没办法理解具体的需求。随着架构的发展，这些指标范围会用来检查系统是否（仍然）符合要求，久而久之，性能指标的变化会反馈出有价值的信息。不过，要找出这些指标范围，并根据它们来检验系统，是件既费时又花钱的事。如果客户不关心系统的性能表现（既没有需求文档，也没有口头要求），也不愿意为性能测试买单，那么性能对他们来说可能不重要，这时你可以从容地将精力放到其他值得关注的系统问题上去。

正确的需求描述应该像这样："必须在 1500 毫秒内响应用户的输入。在正常负载（定义如下……）的情况下，平均响应时间必须控制在 750~1250 毫秒之间。由于用户无法识别 500 毫秒以内的响应，所以我们没必要将响应时间降低到这个范围以下。"

作者简介：

基思·布雷思韦特 1996 年开始写软件赚钱，在此之前他只是个业余爱好者。他的第一份工作是维护用 lex 和 yacc 开发的编译器，此后他设计过 GSM 网络的微波传播模型，用 C++解决过航空运输由于季节性需求变化引起的资费调整问题。随后他从事咨询工作，接触到 CORBA、EJB 和电子商务。目前他是 Zhhlke 公司的首席顾问，管理公司的敏捷实践中心。

一行代码比五百行架构说明更有价值

One Line of Working Code Is Worth 500 of Specification

艾利森·兰德尔（Allison Randal）



设计拥有无穷的魅力。我们运用系统的方法，详细地描述问题空间（problem space），审视解决方案，找出缺陷和可以完善的部分，获得的效果有时令人拍案叫绝。架构说明书（specifications）很重要，因为它描述了构建系统的模式。但是静下心来全面彻底地理解架构——既从宏观上把握组件之间的交互，又着眼于组件内部的代码细节——也很重要。

不幸的是，架构师往往容易被抽象的架构所吸引，沉迷于设计过程。事实上，仅有架构说明书是远远不够的。软件项目的最终目标是建立生产体系（production system），架构师必须时刻关注这个目标，牢记设计只是达成目标的手段，不是目标。摩天大楼的建筑师如果一味追求美观而无视物理定律，迟早会自食苦果。我们的目标是可工作的代码，对软件项目而言，忽略这一点就是灾难。

应该重视团队成员的意见，是他们在实现你的设计。要善于倾听，如果大家对设计提出疑问，很可能设计确实存在问题，或者不够清晰。这时架构师应该与团队成员合作，共同作出决策，修改设计以符合实际情况。没有天生完美的设计，所有的设计都要在实现的过程中逐步完善。

如果你亲自参与开发，应该珍视自己花在写代码上的时间，千万别听信这会分散架构师精力的说法。参与项目所付出的努力，既能拓展你的宏观视野，也能丰富你的微观视界。

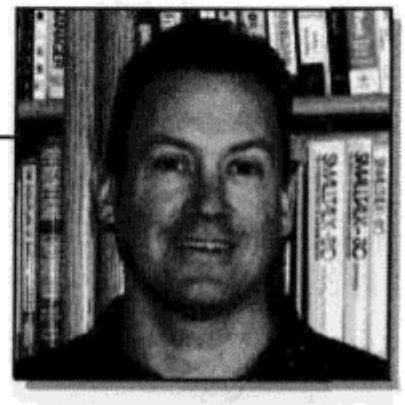
作者简介：

艾利森·兰德尔是开源项目 Parrot 的首席架构师，兼开发组长。在长达 25 年的程序员生涯里，她开发过各种软件，包括游戏、语言分析工具、电子商务网站、编译器、数据库复制系统等；她设计过编程语言，组织过技术交流会，做过项目经理、编辑和顾问。此外还担任过开源软件组织的主席，编写过两本书，并且成立了一家技术出版公司。

不存在放之四海皆准的解决方案

There Is No One-Size-Fits-All Solution

兰迪·斯塔福德（Randy Stafford）



架构师应该坚持培养和训练“情境意识”（contextual sense）——因为我们遇到的问题千差万别，不存在放之四海而皆准的解决方案。

“情境意识”这个贴切的说法，最早由埃贝哈特·雷克廷（Eberhardt Rechtin）在《Systems Architecting: Creating & Building Complex Systems 》（Prentice Hall出版社）一书中提出，并予以深刻阐述：

> [运用“试探法”设计复杂系统架构的要点是]调查有经验的架构师处理复杂问题的方式。有经验的架构师和设计师的答案如出一辙：只须使用常识……[一个]比“常识”更贴切的说法是“情境意识”——在给定情境下对合理性的把握。架构师通过学习和实践，不断积累的案例和经验，建立足够的情境意识。他们通常需要十年的磨练，才能解决系统层次的问题。

在我看来，软件行业的一个大问题，是那些负责解决问题的人积累的情境意识不够。毕竟软件行业起步不过六十多年，并仍在飞速发展。当这个问题消失时，也许就标志着软件行业成熟了。

我做咨询工作时频繁碰到这类问题。典型的情况包括：该用领域驱动设计（注 1）时没有使用；软件方案设计过度，偏离了实用性和眼前的基本需求；或者在解决性能问题时，提出的建议不合理，甚至毫不相关。

掌握软件开发模式的重点在于拿捏应用的时机，这也是分析问题时避免胡乱猜测和矫枉过正的关键。显然，无论是设计系统架构还是分析问题，都不存在万能钥匙，架构师必须培养和训练情境意识，才能更好地设计架构和解决问题。

作者简介见第 11 页。

注 1：见埃里克·埃文斯（Eric Evans）的著作《Domain-Driven Design: Tackling Complexity in the Heart of Software》（Addison-Wesley Professional 出版社）。

提前关注性能问题

It's Never Too Early to Think About Performance

丽贝卡·帕森斯（Rebecca Parsons）



商业用户的需求主要表现为对功能的要求。系统的非功能特性则由架构师负责，包括：性能表现、灵活性、持续正常工作时间、技术支持资源等。但是，对非功能特性的初始测试往往被拖到开发周期的最后阶段，有时还由开发团队来操刀，这样的错误屡见不鲜。

造成这种现象的原因有很多，有人觉得在还没有实现客户要求的功能之前，考虑系统的响应速度和灵活性无异于纸上谈兵；或者面对复杂的环境和测试望而却步；再不就是觉得产品的早期版本不会承担太重的工作负荷。

但是在项目周期的最后阶段才关注性能问题，会导致我们错失大量历史信息，这些信息包含性能变化的细节。如果性能是架构设计的重要指标，就应该尽早展开性能测试。在采用敏捷方法开发的项目中，如果以两周为一个迭代周期，我认为性能测试的开始时间最迟不能晚于第三次迭代。

为什么要提前展开测试？首先，如果性能表现大幅下滑，你至少能找到下滑是由哪些变化引起的。当系统出现性能问题时，你只须检查最近的变化，而不用全盘考虑整个架构。尽早反复地开展性能测试可以缩小问题的可疑范围。

项目伊始的测试数据虽然不能用于性能诊断，但它们至少提供了一个起始基准。这些趋势数据将为今后诊断和解决性能问题提供重要依据。

这样做还可以验证架构和设计是否符合实际性能要求，尤其是对性能要求苛刻的系统，验证的早晚直接关系到能否及时交付项目。

众所周知，坚持技术测试是需要耐心和毅力的，无论是搭建合适的测试环境，采集适当的数据集，还是编写必要的测试用例，都须要投入大量的时间。提前开展性能测试，能让你有条不紊地逐步完善测试环境，为解决性能问题节省下大量的时间和精力。

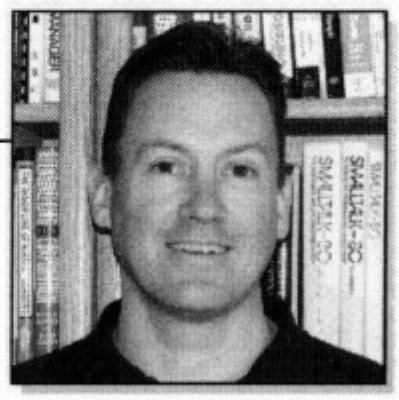
作者简介：

丽贝卡·帕森斯博士是 ThoughtWorks 的首席技术官，拥有 20 多年的应用开发经验，涉及的领域从传统的电信行业到新兴的互联网服务。她在编程语言和人工智能方面著作颇丰，曾服务于多个学术委员会，并仍参与多份期刊的审稿工作。她在建立大型分布式对象应用和整合异构系统方面有着丰富的经验。

架构设计要平衡兼顾多方需求

Architecting Is About Balancing

兰迪·斯塔福德（Randy Stafford）



平衡兼顾各方的要求和项目的技术需求

Balance Stakeholders' Interests with Technical Requirements

提到软件架构，我们脑海里首先浮现的是传统的技术工作，比如系统建模、定义接口、划分功能模块、套用模式，以及优化性能等。此外架构师还要考虑系统的安全性、易用性（usability）、产品支持、发布管理、部署方式等问题。架构师解决这些技术问题和流程问题时，必须考虑相关各方（stakeholder）的要求和利益。只有充分考虑相关各方的要求，才能确保需求说明书的完整性。

架构师要实现的一组最终目标可以通过逐步分析相关各方的需求得到。这个分析过程应该贯彻到软件开发的整个过程中。设计软件架构就是要根据具体情境（context at hand）平衡兼顾这组目标，既满足短期要求，又符合长远需要。

以开发 SaaS（software-as-a-service）业务的部门为例。业务部门的目标包括履行合同义务、创造收益、树立客户口碑、控制成本，以及创造有价值的技术资产，等等。但这些业务目标到了技术部门那里，则转化为：确保软件的功能、

质量和相关品质指标，保证开发团队的工作效率，确保开发的可持续性和可监督性，还有软件产品的寿命和适应性等技术目标。

架构师不仅要为用户创造实用优质的软件，还要运用自己的专业能力，平衡兼顾不同部门的目标，例如 CEO 要求控制成本，运营部门要求软件易于管理，二次开发人员要求代码容易学习方便维护，等等。

有时为了完成紧急任务，架构师可以暂时打破这种平衡，但是要想出色地完成工作，最好维持一种长期稳定的平衡关系。即使打破平衡，也要结合具体情境考虑多种因素，包括估计软件的生命期和商业价值，考虑公司的技术文化和财务惯例，等等。

总之，设计软件架构不是简单技术工作，还要平衡兼顾项目的技术需求和相关各方的业务需求。

作者简介见第 11 页。

草率提交任务是不负责任的行为

Commit-and-Run Is a Crime

尼克拉斯·尼尔森（Niclas Nilsson）



傍晚时候，团队正忙于完成本次迭代的收尾工作，一切按部就班、有条不紊。只有约翰赶着赴约有些急躁，他仓促写完自己的代码，编译、检入，然后匆匆离开。几分钟后红灯亮起（译注 1），构建失败了。约翰没来得及执行自动测试就草率地提交了任务，连累大家无法继续工作。正常的工作秩序全被打乱了。大家清楚，如果现在从版本控制系统中获取更新，得到的将是有缺陷的代码。产品演示迫在眉睫，今天集成的任务还很重，这下计划全乱了。约翰"成功地"破坏了开发流程，在撤消他修改的内容之前，集成工作不得不暂停。

类似的情景反复出现，屡见不鲜。草率提交任务会破坏正常的工作流程，是不负责任的行为。这是开发人员节省时间的常用伎俩，但最终浪费的是他人的时间，性质恶劣。然而这种情况随处可见，为什么呢？因为大家觉得按部就班地构建系统、执行测试太费时间。

碰到这种情况，架构师就该发挥作用了。也许你已经费尽心思设计出灵活实用的架构，也说服大家采用了包括测试驱动开发（Test-driven Development）在内的敏捷方法，并且搭建了用于持续集成的服务器。你还要营造一种团队文化：

译注 1：许多采用敏捷开发方法的软件公司（例如 ThouthtWorks）在每个团队成员的桌上放置一盏三色灯，用来指示当前的集成状态，如果指示灯显示黄色，表示正在集成；如果显示绿色，表示上一次集成通过，开发人员在这时候获得的代码是可用而可靠的；如果显示红色，则代表集成失败。

以维护流程通畅为重，以浪费他人时间为耻。要做到这一点，务必在系统内实现完善的自动测试功能，纠正开发人员的行为。如果测试简单方便，开发人员会更乐于执行，这本身大有裨益，同时避免把有缺陷的代码丢给同事。如果测试依赖于外部系统，或者须要访问数据库，则有必要重新设计使其可以在本地完成，例如采用 Mock 或 Stub 的测试方法，或者利用运行在内存中的数据库（in-memory database），在构建服务器上慢慢执行。人们不愿等待计算机，如果被迫等待，他们就会走捷径，结果常给别人造成麻烦。

沉下心来改善系统的生产效率，缩短流程，避免各行其是，才能缩短开发时间。采取一切可行的措施，例如运用模拟方法、降低依赖性、细致划分系统模块，等等。总之要杜绝一切草率提交任务的念头。

作者简介：

尼克拉斯·尼尔森身兼软件开发教练、顾问、教育工作者及作家等多个身份，他热心软件开发工艺，醉心于精良的设计与架构。他是 factor10 公司的创始人之一，同时还是 InfoQ 架构社区的首席编辑。

不要在一棵树上吊死

There Can Be More Than One

基思·布雷思韦特（Keith Braithwaite）



负责构建系统的人似乎无法接受这样的事实：没有哪种数据模型、消息格式、消息传送机制，甚至主流的架构组件、策略、观点能够单独用来解决所有的业务问题，毕竟大家都希望摆脱业务需求不断滋生的意外和烦恼。但事实如此，如果企业规模庞大，业务复杂，即便是设置“账户”表，也必须给出多种不同的设计，才能适应未来 10 年公司的发展，单一的“账户”表无论如何也不可能满足这种需求。

在技术领域实现唯一性相对容易。但在业务领域，大家要面对千头万绪、纷繁复杂的现实世界。更糟的是，业务压根就不是与“现实世界”打交道，而是要应付人们对局部世界片面的看法和意见。有人把业务领域的问题技术化，妄想找到唯一解。但“现实”正如菲利普·迪克（译注 1）笔下描述的，“一旦发现事实，就再也无法回避”（that which does not go away when one stops believing in it）（译注 2）。随着业务发展，麻烦迟早会出现。于是企业数据团队之类的组织应运而生，他们浪费宝贵时间争执不休，妄想借助文档类型定义（DTD，Document Type Definition）的方法治疗存在性忧虑症（Existential Dread）（译注 3），不过是水中捞月罢了。这类服务很难令付费客户满意。

译注 1：菲利普·迪克（Philip Kindred Dick），美国当代著名科幻小说作家，生前穷困潦倒，屡遭退稿，身后被世人重视，作品多被搬上影幕，包括：《银翼杀手（Blade Runner）》、《全面回忆（Total Recall）》、《少数派报告（Minority Report）》、《盲区行者（A Scanner Darkly）》、《冒名顶替（Impostor）》等。

译注 2：出自迪克的小说《银翼杀手》，原指男主角开始怀疑自己是克隆人，念头挥之不去。这里借喻现实世界的复杂多变是不可避免的。

译注 3：心理学术语，指一种根本性焦虑症，这里指面对复杂多变的现实世界时不知所措的心态。用定义文档的方法治疗心理疾病显然荒唐，这是作者的讽刺。

为什么我们不愿承认世界是混乱的？为什么不接受服务方式、表现形式、解决方案可以是多样的、不一致的和重叠的？因为大家都怕遇到错综复杂的依赖关系，担心数据更新无法同步，还有额外的维护开销，这些都是技术人员的噩梦。不妨看看数据仓库给我们的启示。概要性质的数据集市（译注 4）通常都是非规范化的，导入数据比较自由，计算也相对随意，展现出来的数据视图与底层的数据库大相径庭。但是数据集市仍然可以正常工作，没有因为这些非功能属性（译注 5）而失效。因为 ETL 过程（译注 6）可以将事务处理和分析处理两个截然不同的世界联结起来，哪怕两者在更新频率、查询频率、吐吞量、设计的变更频率，甚至容量上有着迥然不同的要求。关键在于子系统的设计上，要充分利用非功能属性的差异性，实现对不同表现形式的管理。

采用多种表现方式、多种传输方式不是为了消遣。应当认识到，通过分解系统的非功能参数，可以为客户提供多样化的解决方案。

作者简介见第 21 页。

译注 4：数据集市（data mart），可以理解为数据仓库的某个子集，其数据模型是从数据仓库抽象出来的，用以满足特定部门的应用需求，数据集市可以在一定程度上缓解访问数据仓库的速度瓶颈。

译注 5：组件式软件服务包括功能属性（functional properties）和非功能性属性（nonfunctional properties）。简单来讲，功能属性是指系统所能提供的功能，而非功能性属性指这些功能的实现方式。

译注 6：ETL 过程（ETL Process）是数据抽取（Extract）、转换（Transform）、装载（Load）的统称，它负责将数据源向目标数据仓库转化，是构建数据仓库的重要环节。

业务目标至上

Business Drives

戴夫·缪尔黑德（Dave Muirhead）



在商业化的背景下开发企业应用，架构师必须成为业务部门和技术部门之间沟通的桥梁，周旋调解，兼顾双方的利益，同时用业务目标来驱动项目开发。业务目标和实际的开发条件应该成为架构师主持制定决策的参照系统。

按照通常的业务惯例，在启动一个软件项目之前，应当制定计划，明确对投资回报率的预期。架构师必须把握这个预期，并估计该项目的商业价值，避免作出错误的技术决策，造成经费超支。每当要权衡取舍时，无论是与业务部门讨论是否应该实现某项功能，还是与开发团队讨论技术上的设计与实现，都应该把高投资回报率当作目标。举例来说，架构师必须谨慎地站在业务团队一边，拒绝开发团队选用价格不菲的软件和售后服务成本过高的技术。

如何提供足够的高质量信息，持续反馈开发中的成果，用于支持业务决策，是用业务目标驱动项目开发的难题之一。关键在于加大透明度。架构师必须与开发团队合作，设法建立反馈回路，持续有规律地向业务部门提供信息。精益软件开发（lean software development）的许多技巧可以借鉴，比如说，使用直观

的大图表，采用持续集成，以及在项目开始后尽早地、频繁地发布可工作的软件，等等。

总的来说，软件开发是一种设计工作，因为在形成产品的过程中需要持续地制定决策。软件开发人员可以制定（技术）决策，但通常不该参与业务决策。业务部门应该为技术部门提供指导，解答他们的问题，并制定业务决策。如果业务部门不能履行这些职责，相当于把业务决策的任务甩给了开发人员。架构师必须通过沟通协调，既保护软件架构，又坚持业务目标，既允许开发人员制定微观（技术）决策，又设法避免他们参与制定业务决策。如果技术决策脱离了业务目标和现实条件的约束，则无异于用宝贵的稀缺资源进行高风险的投机。

用业务目标驱动项目开发，才能保证软件开发团队的长远利益。

作者简介：

戴夫·缪尔黑德是位身经百战的老程序员，同时还是位业务技师（business technologist）。他是 Blue River Systems Group 有限公司的老板，兼任公司的首席顾问。这是一家总部设在丹佛，提供技术策略咨询的敏捷软件开发公司。

先确保解决方案简单可用，再考虑通用性和复用性

Simplicity Before Generality, Use Before Reuse

凯佛林·亨尼（Kevlin Henney）



许多用来实现基础设施的代码，包括组件框架、类库、基础服务（foundation services），普遍存在一个问题，它们的设计一味强调通用性而不考虑具体应用，导致出现许多令人困惑的可选项和不确定因素，这些功能常常不是被闲置，就是被误用，甚至毫无价值。多数开发者开发的是专用系统，无限制的通用性对他们的帮助不大。寻求通用性最好的办法是研究现有的具体案例，抓住问题的实质，从根本上得出通用解决方案。通过经验提炼的简单方案，远胜过不切实际的通用性。

如果存在多个可实施方案难以取舍，“先简单后通用”原则可以成为最终的评判标准。挑选基于具体需求的简单方案，放弃鼓吹通用性的复杂方案。而且简单的方案在实践中完全有可能表现出更好的通用性。退一步来说，修改简单方案满足需求，也比修改通用方案容易，因为通用方案常常在最关键的地方使不上劲儿。

虽然许多通用设计的初衷是好的，但还是难逃失败的命运。设计组件的首要任务是抓住具体需求，满足需求。通用性来自对需求的理解，理解之后才能简化。

提炼通用性可以使我们更加接近问题的本质，通过分析已有案例可以获得清晰、简洁、有依据的规律和方法。然而提炼通用性往往流于形式，南辕北辙，不但无法减少复杂性，反而增加复杂性。追求理论上的通用性通常会导致解决方案脱离实际的开发目标。由于这种通用性基于错误的假设，所以无法提供有价值的方案，只会带来棘手的问题，增加开发人员和架构师将来必须面对的偶发复杂性（accidental complexity）。

虽然很多架构师重视通用性，但这样做是有前提条件的。并非所有人都需要通用性，愿意为它掏钱，具体情况要具体分析，有针对性的解决方案才有价值。我们提供具体解决方案时，无须排斥通用性和灵活性，但是如果过早脱离具体情况，只会迷失在无限的可能性里，被复杂的配置选项、超负荷的参数列表、冗长啰嗦的接口，以及存在缺陷的抽象所淹没。追求随心所欲的灵活性，会使人们在无意中错失（有些人甚至故意忽略）更简单的设计和更有价值的特性。

作者简介：

凯佛林·亨尼是位独立顾问/培训师。他关注设计模式和架构、编程语言和技巧，以及开发过程和方法。他曾参与撰写两本书，分别是《Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing》和《Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages》（均由 Wiley 出版公司出版）。

架构师应该亲力亲为

Architects Must Be Hands On

约翰·戴维斯（John Davies）



称职的架构师应该通过示范领导团队。他能胜任团队的所有工作，从网络布线到配置构建流程，从编写单元测试到担任测试工作。对技术缺乏全面理解的架构师，充其量只是个项目经理。团队成员通常具备深厚的专业知识，很难想像不懂技术的架构师如何赢得大家的信任。众所周知，架构师是业务团队与技术团队之间的接口人，他必须理解各种技术问题，无须频繁求助他人，才能代表技术团队发言。同样，架构师还要懂得业务知识才能督促技术团队满足业务需求。

架构师就像航班的主驾驶员，看起来不是很忙碌，但他经验丰富，持续地监视着情况，一旦发现异常随时采取行动。项目经理（副驾驶员）负责日常的管理工作，将架构师从烦琐的杂务和人事管理中解脱出来。架构师对项目的交付和质量负有最终责任，没有威信很难展开工作，威信与项目的成败密切相关。

人们擅长通过观察来学习，我们打小就是这样。优秀的架构师应该有能力发现问题所在，召集团队成员，向大家解释问题产生的原因，或者给出巧妙的解决方案，而不是寻找替罪羊。架构师完全可以要求团队成员的帮助，让团队成员

充分参与制订解决方案，同时引导讨论方向，找出正确的方案。

架构师应该尽早参与项目，与团队成员并肩工作，而不是坐在象牙塔里发号施令。不能把技术决策和方向上的难题拆分出来扔给别人，或另辟新项目来解决，应该采取更务实的办法，比如亲自动手研究，或者咨询同行的意见——优秀的架构师之间都保持着紧密的联系。

称职的架构师应该至少熟练掌握一种专业工具（例如一种 IDE），他们应该身先士卒。按道理说，软件架构师应该会用 IDE，数据库架构师应该会用 ER 工具，信息架构师应该会用 XML 建模工具。而一位技术/企业架构师，起码应该熟练运用各个层次的工具，从使用 Wireshark 检测网络流量，到利用 XML Spy 给复杂的财务信息建模——无论简单还是复杂的都该掌握。

架构师通常都取得过不错的业绩，有份出彩的简历，容易获得业务人员和技术人员的青睐。但除非他能展示自己的实践能力，否则很难赢得团队的尊重，团队成员将无法从他身上学到东西，大家甚至难以在他的带领下做好本职工作。

作者简介：

约翰·戴维斯目前是美国 Revolution Money 公司的首席架构师。他最近创办了一家新公司，取名 Incept5。

持续集成

Continuously Integrate

大卫·巴特利（David Bartlett）



构建应用程序作为重大项目事件的日子已经一去不复返了。架构师（无论是应用架构师还是企业架构师）都应该在项目中鼓励推广持续集成的方法和工具。

持续集成的说法最早是由马丁·福勒（Martin Fowler）在设计模式中提出来的，它指一套频繁对应用程序进行自动化测试和构建的实践方法，以及确保测试和构建自动执行的相关工具。持续集成通常在一台专门配置的集成服务器上完成。得益于单元测试工具和方法的发展，加上自动化构建工具的配合，持续集成已经成为当今软件开发中必不可少的手段。

持续集成针对的是所有软件开发过程中的一个共同阶段：将源代码转换成可运行应用程序的过程，在这个过程中，把众多开发成果汇集起来进行测试。你以前可能听过“尽早构建，经常构建”（build early and often）的提法，它确保当前的开发不会出现意外，是一种降低风险的技巧。现在持续集成已经取代了“尽早构建，经常构建”。除了构建应用程序，持续集成还包括增进团队沟通与协作的特性。

构建应用程序是持续集成最主要的内容，它通常是自动执行的，可以设置在夜里进行，或者当源代码变更时自动触发。当然你也可以选择手动构建。构建开始后，先从版本库取出最新版本的源代码，持续集成工具会尝试对项目进行构建，然后展开测试。结束后，系统会发出相应的通知，报告集成结果。通知可以通过多种途径发送，包括电子邮件和即时通信工具。

架构师青睐持续集成，因为借助它可以取得更稳定、更符合要求的开发成果。更重要的是，它可以提高公司和开发团队的工作效率，改善工作效果。

作者简介：

大卫·巴特利是位对编程充满热情的专业程序员，拥有超过 25 年的开发经验，身兼开发者、架构师、项目经理、技术顾问和培训讲师等多种身份。他开办了一家名为 Commotion Technologies 的私人公司，为客户提供咨询服务，同时还在宾夕法尼亚州立大学大峡谷分校研究生工程学院授课。他目前的主要工作包括与费城联邦储备银行合作，设计网页、搭建门户网站，同时开发组合式应用供联邦储备系统和美国财政部使用。

避免进度调整失误

Avoid Scheduling Failures

诺曼·卡诺瓦利（Norman Carnovale）



导致项目失败的原因很多，最常见的是中途临时调整进度。要保证调整后的进度，只能靠大伙加班加点。当然，调整也可能指延长项目期限，或者增加项目资源，那就没什么好操心的了。最怕的是时间不变，任务量增加；或者任务不变，截止日期提前。

一般人有一种错误的观念，认为加快进度可以降低成本，提高交付速度。为了缩短交付时间，开发人员常常被要求加班，甚至放弃“不太重要的计划任务”（例如单元测试）；就算交付时间不变，也可能被要求增加额外的功能。架构师应该不惜一切代价拒绝这类要求，提醒那些提出要求的人，改变计划会带来以下问题：

- 仓促决定的进度会导致拙劣的设计、蹩脚的文档，可能引发质量问题，导致用户拒绝验收；
- 仓促完成的代码，会直接导致最终产品的 Bug 数量增加；
- 紧张的测试进度会导致测试不充分，直接增加测试中可能出现的问题；
- 以上几项都会引发产品质量问题，而解决产品质量问题的代价更高。

最后的结果是成本不降反升，通常项目就是这样失败的。

作为架构师，你难免会遇到类似的情况，为了确保项目顺利进行，应该迅速采取行动表明立场。首先通过协商尽量维持原定进度，保证产品质量；如果必须加快进度，可以尝试去掉一些不重要的功能，留待后续版本发布。显然这需要提前做好准备，包括谈判策略和说服他人的技巧。不妨从今天开始培养这些方面的能力，关键时一定会派上用场。

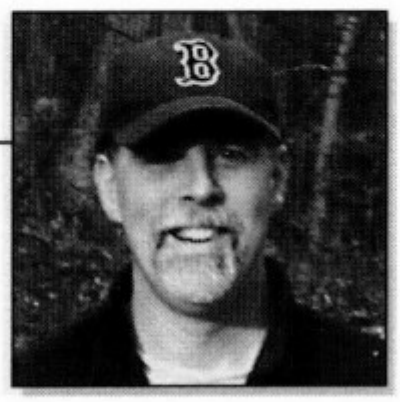
作者简介：

诺曼·卡诺瓦利是洛克希德马丁公司的 IT 架构师，负责为国土安全局相关项目提供专业技术服务。他做过软件顾问、技术讲师，曾经就职于 Davalen 公司（*http://www.davalen.com*），该公司是 IBM 的重要业务伙伴，专门从事 WebSphere Portlet Factory、WebSphere Portle 和 Lotus Domino 相关的项目开发。

取舍的艺术

Architectural Tradeoffs

马克·理查兹（Mark Richards）



架构师应该明白鱼和熊掌不可兼得的道理。世上不存在十全十美的设计——既具有高性能，又具有高可用性；既高度安全，又高度抽象。有一个真实的历史事件，软件架构师应该烂熟于心，在与客户或同事沟通时能派上用场。这就是瓦萨号（Vasa）战舰的故事。

17 世纪 20 年代，瑞典和波兰之间爆发战争（译注 1）。瑞典国王（译注 2）迫于战争经费的压力，急欲速战速决，他下令建造一艘名为瓦萨号的战舰。这可不是一艘普通的战舰，其设计要求绝非同时代战舰可比：舰长超过 60 米，两个炮台上配备 64 门舰炮，可以将 300 名士兵从水路安全运送到波兰。时间紧迫，资金紧张（类似的情况架构师可能都经历过），而且设计师从未设计过这种规模的战舰（他擅长设计单炮台的小舰艇），只能凭经验和猜测着手设计。最终，工匠们按照设计说明建造了战舰。下水那天，瓦萨号在隆重的仪式中驶入海港，鸣完礼炮后，径直沉入了海底。

瓦萨号的问题很明显。参观过 17、18 世纪大型战舰的人都知道，甲板上空间有

译注 1：17 世纪初，瑞典和波兰曾因争夺波罗的海沿岸的利夫兰而爆发战争。1626 年，双方再次发生军事冲突，瑞典抢占了波兰的波莫瑞地区的大片土地，1629 年，双方签订停战协议。

译注 2：瑞典国王古斯塔夫斯·阿道弗斯，1611~1632 年在位，是瑞典帝国的缔造者，一生穷兵黩武，被称为“现代战争之父”，37 岁时战死沙场。

限而且危险，作战时情况更糟。建造一艘既能作战又能运兵的战舰是一个巨大的错误。设计师为了满足国王的心愿，设计了一艘性能失衡、不堪一击的战舰。

软件架构师应该从这个故事中汲取教训，避免在工作中重蹈覆辙。妄想实现所有需求（像瓦萨号一样），只会产生脆弱的、一无是处的架构。举个取舍的例子，如果要求面向服务的架构（SOA）实现点对点解决方案（point-to-point solution），你就不得不放弃 SOA 方法中的各种抽象层，设计出像意大利面条一样盘根错节的架构。有许多工具可以帮助架构师做出取舍，最流行的是架构权衡分析方法（Architecture Tradeoff Analysis Method，ATAM）和成本收益分析方法（Cost Benefit Analysis Method，CBAM）。如果想了解这两个工具的详细信息，可以访问软件工程协会（Software Engineering Institute，SEI）的网页：*http://www.sei.cmu.edu/architecture/ata_method.html* 和 *http://www.sei.cmu.edu/architecture/cbam.html*。

作者简介见第 9 页。

打造数据库堡垒

Database As a Fortress

丹·恰克（Dan Chak）



所有数据，包括员工输入的数据和从客户那里收集来的数据，都存放在数据库里。用户界面和应用逻辑会变化，业务会发展，人员会变动，但是数据会永远保留下来。所以，创建牢固的数据模型要从第一天开始，这绝非言过其实。

敏捷方法盛行使很多人认为在有需要时才设计应用是可行的，甚至是更可取的。提前展开全面综合技术设计的日子已经成为过去。新派观念提倡尽早地、频繁地部署应用；一行写进产品的代码比头脑中的十行更有价值。这些听起来近乎完美的观点，对数据库却行不通。

尽管业务规则和用户界面经常变化，但是采集来的数据的内部结构和关系通常不会变化。因此，通过正确分析，首先从结构上定义好数据模型非常关键。将数据原封不动地从一种模式迁移到另一种模式是非常困难的，不但耗费时间，而且容易出错。应用层出错还可以暂时忍受，数据库出错则是灾难性的。一旦数据被破坏，即使事后能够修正数据层的设计问题，丢失的数据也无法恢复了。

牢固的数据模型既可以保障当前数据的安全，又为今后提供可扩展性。要保障数据安全，就必须隔离来自应用层的 Bug（在不断变化的应用层中，这些 Bug 无处不在，不会因为你的勤奋而消失）；必须严格遵守引用完整性（referential integrity）规则，尽可能使用域约束（domain constraints）规则；还要选择恰当

的键（keys），既保证数据的引用完整性，又遵守约束规则。要实现可扩展性，就必须正确地将数据标准化（normalizing），以便今后在数据模型上添加架构层；千万不要偷懒走捷径。

数据库是保护珍贵数据的最后一道关卡。应用层的设计经常变动，无法保证自身的安全。为了妥善地保护数据库，数据模型的设计必须做到能够拒绝无效数据，阻止无意义的关系。在定义键、外键和域约束时，应该采用简洁的，容易被理解和验证的名称，使它们的含义不言自明（self-documenting）。数据模型中的域规则也要做到物理化和持久化（physical and persistent），避免它们在应用逻辑发生改变时被删除。

为了充分发挥关系型数据库的作用——让它真正成为应用的一部分，而不仅仅是存放数据的库房——必须从开始构建数据库时，就深刻地理解业务需求。随着产品的演变，数据层也会发生变化，但无论何时，都要确保数据层像堡垒一样坚固。如果你信任数据层，赋予它捕获其他架构层 Bug 的重任，它不会让你失望的。

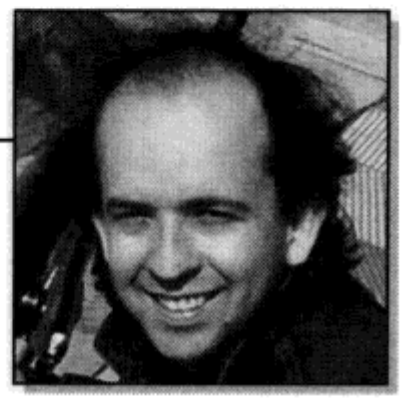
作者简介：

丹·恰克是华盛顿邮报下属的 CourseAdvisor 公司的研发总监，著有《Enterprise Rails》（由 O'Reilly 出出版）。

重视不确定性

Use Uncertainty As a Driver

凯佛林·亨尼（Kevlin Henney）



面临两种选项时，多数人认为最重要的是从中选择，然而设计（无论是软件设计还是其他设计）并不是这样的。当你面对两种可能性时，应该仔细考虑设计中存在的不确定性。不确定性可以促使你推迟决定（译注 1），收集更多的信息；促使你用分隔（partition）和抽象（abstract）的方法来降低设计决策的重要性（译注 2）（significance of design decisions）。如果抱着头脑中最先闪现的想法不放，被它束缚，偶然性决策就会占上风，软件的灵活性会降低。

格雷迪·布奇（Grady Booch）对架构的定义是众多定义中最简单也最具有建设性的："架构属于设计范畴，但并非所有的设计都属于架构之列。架构代表了那些形成系统的重要设计决策。其重要性由变更决策的代价来衡量。"也就是说，优良的架构能够从整体上降低设计决策的重要性，糟糕架构则会突出重要性。

如果出现两个合理的选择，架构师应该停下来，设法找出介于两者之间的、具有更低重要性的决策，而不是简单地在两者中作出选择。了解两者之外还存在其他选择（这个选择并非显而易见），比决策结果本身更有价值。

译注 1：推迟决定（defer commitment）是精益软件开发的原则之一。推迟决定不是故意拖延，而是强调作出的决定应该基于足够的事实，不能仅凭假定和猜测。

译注 2：重要性（significance），指设计决策对系统的影响程度。重要性用变更设计决策须要付出的代价来衡量，如果变更某个决策须要付出的代价高，那么其重要性就高，反之则低。

架构师往往要冥想苦思反复尝试，才能清楚地将问题一分为二。当你积极地与同事在白板前争论不同的可能性时，当你对着代码反复琢磨而无法决定采用哪种实现方式时，当新的需求或对需求的新解释质疑现有实现方式时，说明你碰到不容易确定的情况了，这时要设法利用分离（separation）或封装（encapsulation）将决策和最终依赖于决策的代码隔离（isolate）开。做不到这一点，代码就会杂乱无章，仿佛一个紧张的应聘者，试图用模棱两可、含糊其辞的方式回答没有把握的问题。还有些盲目自信的人会作出武断的选择，但是开快车又不愿回头看是会拐错弯的。

迫于压力，人们常常为了决策而决策。这时可以借鉴期权思想（options thinking）（译注 3）。当你在不同的系统开发路线之间举棋不定时，不要急于作出决策。推迟决策，直到掌握更详实的信息，以便作出更可靠的决策。但也别太迟，要赶在这些信息失效前利用它们。

架构和过程相互交织，所以架构师应该在开发周期中促成那些注重实证的架构方法，并设法引出反馈，建设性地利用不确定性，将系统和进度分割开来。

作者简介见第 37 页。

译注 3：“期权思想”是指在期权交易中，权利的受让人可以在将来某个约定的时间，根据当时的情况决定是否行使权利，即推迟作决定时间。

不要轻易放过不起眼的问题

Warning: Problems in Mirror May Be Larger Than They Appear

戴夫·奎克（Dave Quick）



我参与过数以百计的软件开发项目，无一例外都存在这样或那样的隐患，这些问题常常会导致难以预料的后果。问题出现时，虽然个别团队成员会发现一些端倪，但往往由于大多数人认识不到其严重性，这些问题不是被忽略就是被搁置，直到变得难以解决。

造成这种情况的原因包括：

- 问题刚出现时一般都不起眼，直到后期才会变得严重。温水煮青蛙的实验或许只是个故事，但是用来形容某些项目非常合适。
- 当个人的经验和知识得不到其他团队成员的认同时，你的意见就会遭到抵制。克服这种困难需要极大的勇气、自信和超乎寻常的口才。很少人能坚持力排众议，哪怕是高薪聘来专门解决这类问题的资深顾问。
- 大多数程序员都是乐观主义者。痛苦的经历可以抑制乐观情绪，缺少这类经历则容易导致盲目乐观。天生的悲观主义者往往不受团队欢迎，哪怕他总是对的。在没有十足把握的情况下，没人愿意冒险与多数人作对。如果只是说："我觉得这样做不妥，但说不出理由"，是很难获得别人认同的。
- 每个团队成员关注的侧重点不同。通常大家关心的是个人职责，而不是项目的整体目标。
- 每个人身上都存在自己难以识别和接受的盲点和不足。

下面的方法有助于克服这些消极因素：

- 组织团队一起来想办法管理风险。例如用跟踪 bug 的方法来跟踪风险。让大家都参与识别风险，然后进行跟踪，直到风险解除。为风险划定等级，每当风险状态发生变化，或者有新情况发生时，重新评估风险的等级。这样做可以避免主观因素的影响，同时有助于提醒团队定期重新评估风险。
- 如果你的观点不被大家接受，应该设法寻找更容易让他们理解的表达方式。鼓励大家重视反对意见，寻找更理性的讨论方式。
- 不要轻易放过“不妥”的感觉。如果还没有足够的证据证明“不妥”，请设法寻找最简单的方法来证明。
- 多和客户交流，经常与团队沟通，看看你是不是真的了解他们的想法。用户需求记录优先级列表（a prioritized list of user stories）之类的工具虽然可以帮助你完成工作，但是无法替代定期与客户沟通的作用，你更需要的是开放的思想。
- 自己的盲点自己难以查觉。忠言虽然逆耳，却是你最宝贵的财富。

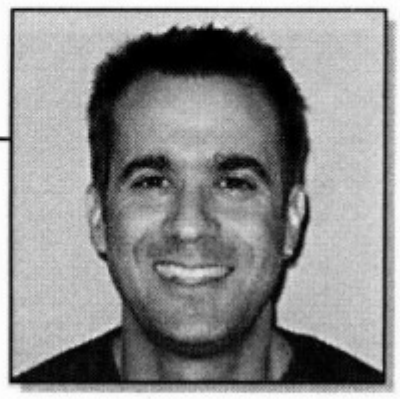
作者简介：

戴夫·奎克是 Thoughtful Arts 公司的老板，也是该公司唯一的员工，同时兼任公司的首席架构师和门卫。Thoughtful Arts 为音乐工作者提供定制的音乐软件，也为开发音乐和艺术类软件的公司提供设计咨询服务。

让大家学会复用

Reuse Is About People and Education, Not Just Architect

杰里米·迈耶（Jeremy Meyer）



有这样一种观点，认为设计优良的框架、细致考虑并精巧实现的架构自然会被人们重复利用。事实上，即便是最精美的架构、最优雅的框架、可复用性最高的系统，也必须满足下面的条件才可能被复用。

大家知道它们存在

Know it's There

为了让公司内的开发人员和设计人员重复利用已有的设计、框架、函数库或代码段，必须先让他们知道这些资源的存在，以及在哪里可以找到相关的信息（比如文档、版本和兼容性等）。这里的逻辑很简单：人们不会寻找不知道的东西。当大家可以轻松获取这些信息时，复用的几率就会上升。

在公司内推广可复用资源的办法有很多。规模较大的团队可以通过 Wiki 页面和 RSS 订阅来更新信息，或者利用 E-mail 通知大家版本库的更新情况。在小团队里，设计师和开发组长可以逐个通知同事，或者干脆大喊一嗓子，通知办公室里所有的人。总之，你必须有办法推广可复用的资源，无论采用哪种形式都好过置之不理。

大家知道如何使用它们

Know How to Use It

掌握如何利用已有的资源需要一定的技巧和培训。当然，某些天才的开发人员和架构师（按照 Donald Knuth 的说法）能够和代码、设计产生“共鸣（resonate）”，他们的理解能力和理解速度令人印象深刻，甚至可怕。但这种人毕竟罕见，团

队里余下的那些称职的、严谨的、有领悟能力的开发人员和设计人员仍然需要这方面的训练。

开发人员和设计人员有可能不明白设计中采用的特定设计模式，或者无法透彻地理解框架设计者希望他们使用的继承模式，应该为他们提供方便获取相关信息的途径，例如提供最新的文档，最好能开展培训。简单的培训可以获得很好的效果，使大家对复用保持一致的认知。

大家认识到利用已有资源好过自己动手

Are Convinced That It's Better Than Doing It Themselves

一般人（特别是开发人员）倾向于自己解决问题，不愿意寻求别人的帮助。他们觉得向别人请教问题是一种浅薄，甚至无知的表现。“好过自己动手”对不同的人有不同的涵义，与个性类型，以及心智是否成熟有很大关系。刚参加工作的人凡事喜欢自己动手，展现个性。但有经验的员工更乐意了解别人对类似问题的思考，希望最好已经有了解决的办法。

如果大家找不到可复用的资源，或者不知道如何使用这些资源，人的天性就会发挥作用：他们会自己动手实现。到头来吃亏的还是架构师。

作者简介：

杰里米·迈耶有近 20 年设计软件、开发软件，以及教授相关课程的经验。他目前是 Borland 软件公司的首席顾问，从事建模和设计方面的研究。

架构里没有大写的“I”

There Is No 'I' in Architecture

戴夫·奎克（Dave Quick）



英文单词架构（architecture）里有字母“i”，但不是大写字母“I”。它代表的不是那个喜欢唤起别人关注，喜欢凌驾于众人之上的“I”（自我）。小写字母“i”放在这里很适合，因为它符合正确的拼写和发音要求。

这和架构师有什么关系呢？自我可能是我们最大的敌人。相信大家都遇到过这样的架构师：

- 他们认为自己比客户更懂需求。
- 他们认为开发人员只是雇来实现自己想法的资源。
- 如果他们的想法遭到质疑，或者旁人指出他们忽略了他人的意见，他们会极力为自己辩解。

我猜凡是有经验的架构师都犯过类似的错误。因为这些错误我都犯过，而且教训惨痛。

为什么我们会犯这样的错误？

- 我们取得过优秀的业绩。成绩和经验促成了我们的自信，使我们成为架构师，也让我们有机会接触更大的项目。在自信和自负之间有着一条非常微妙的界线。有时项目的要求超出了个人能力的范围，我们不自觉地越过了这条界线，自负就趁虚而入了。
- 大家尊重我们。复杂的设计问题构成的技术壁垒，使我们免于遭受批评，但是保守、自负和对经验的倚重会导致设计上的疏漏。
- 架构师也是普通人。每项设计都凝聚了我们的心血。如果大家批评你的作品，你会觉得是批评你本人。辩解容易，难的是学会停止辩解；恃才傲物容易，难的是拥有自知之明。

如何避免犯这样的错误？

- 需求不会撒谎。面对完整无误的需求，任何人只要将其实现，都是称职的架构师。应该与客户密切合作，确保双方理解每项需求的业务价值。驱动架构的是需求，不是架构师，你的任务是竭尽所能满足需求。
- 重视团队合作。同事不仅是资源，也是你的设计伙伴和安全网。不被赏识的人是不会勤奋工作的。架构属于团队，不是你一个人的。你负责导航，大家一起划桨。双方缺一不可，但相比之下，你更离不开他们的帮助。
- 检查你的工作。你的模型只是你对架构的理解，不一定是最合适的架构。应该和团队一起决定测试方法，检查架构对每项需求的支持情况。
- 自我反省。偏袒自己的成果，关心个人利益，认为自己最聪明，这些都是人类的天性。这些缺点在压力下更容易暴露。每天应该花几分钟反省自己的处事方式：是不是尊重每个人提出的想法，并表示了感谢？有没有否定善意的建议？是不是真的理解别人为什么不赞同你的做法？

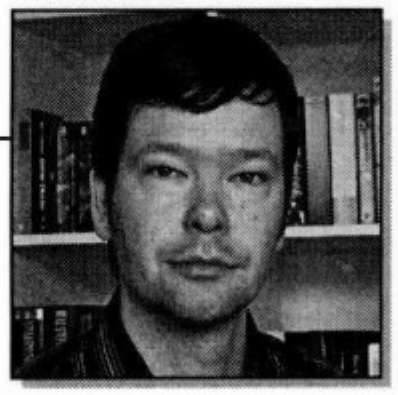
去掉架构中的大写“I”并不能保证成功，但它可以杜绝常见的，因为强烈的“自我意识”引发的问题。

作者简介见第 51 页。

使用“一千英尺高”（译注 1）的视图

Get the 1,000-Foot View

埃里克·多伦伯格（Erik Doernenburg）



架构师都希望了解正在开发的软件质量如何。软件质量的外在表现是满足客户的需求，其内在表现则比较隐蔽，包括设计是否清晰，是否容易理解、维护和扩展。如果被人追问质量的定义是什么，我们通常只能敷衍道：“只要看到，我就知道”，可是我们怎么才能看到质量呢？

在架构图里，系统是由若干个小方框组成的，方框之间的连线代表着各种含义：依赖关系、数据流、共享资源（例如总线）等。这种图好比从飞机上俯瞰地面风景，我们称为“三万英尺高”（译注 2）的视图。另一种典型的视图是源代码，好比站在地面上看大地。两种视图都无法充分展现软件的质量：前者太抽象，而后者细节太多，以致我们看不清整个架构。很显然，我们需要一个介于两者之间的视图——“一千英尺高”的视图。

“一千英尺高”的视图提供的信息来自恰当的层次，囊括了大量数据和多种度量标准（multiple metrics），例如方法数（method count）、类扇出数（class fan out）（译注 3）和圈复杂度（cyclomatic complexity）（译注 4）。具体的视图与特定的质量属性密切相关，例如可视化的依赖关系图、在类的级别上显示多种度量标准的柱状图，以及复杂的、关联多个输入值的复合标准视图（polymetric view）。

译注 1：一千英尺≈305 米。

译注 2：三万英尺≈10000 米。

译注 3：类扇出数（class fan out）。扇出数（fan out）原是半导体电路中的概念，用于表示输出逻辑门可驱动同类型输入逻辑门的数量。类扇出数则是用来描述类之间的耦合度的一种代码度量，其数值表示一个类依赖的其他类的个数。

译注 4：圈复杂度（cyclomatic complexit），又称为条件复杂度（conditional complexity），它通过测量源代码中的线性无关路径数来衡量程序的复杂度。

纯粹靠手工绘制这些视图，并且保持它们与软件同步是不现实的，我们可以借助工具直接根据源代码创建视图。虽然有专门绘制视图（例如设计结构矩阵）的商业工具包，但是利用提取数据的小工具，加上通用的绘图工具包，也可以非常容易地绘制出想要的视图。举一个简单的例子，可以将 Checkstyle（译注 5）的输出结果（一组针对类和方法的度量标准及结果）导入电子表格工具，然后生成图表。Checkstyle 的输出结果还可以使用 InfoViz（译注 6）工具箱绘制成树状图。GraphViz（译注 7）则是绘制复杂依赖关系图的理想工具。

一旦我们绘制出合适的视图，判断软件质量就更客观了。借助视图，可以将开发中的软件和若干相似的系统进行比较。对比软件的不同版本可以显示变化的趋势，对比不同子系统的视图可以发现异常之处。即使只有一张图表，我们也可以依靠技巧发现其中的模式和美学规律。对称性良好的树状结构很有可能代表着合理的类层次结构，比例协调的框图可能是合理分配类大小的表现。许多时候，外表和内在是一致的。

作者简介：

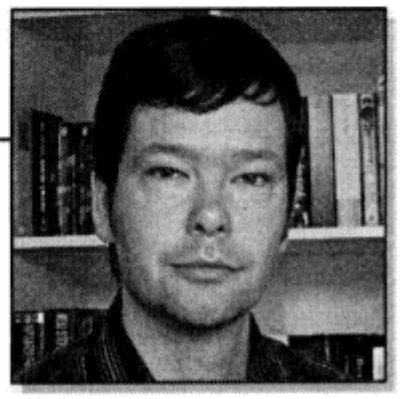
埃里克·多伦伯格是 ThoughtWorks 公司的技术专家，负责大型企业解决方案的设计和实现。

译注 5：Checkstyle 是一款检查 Java 代码规范的软件。
译注 6：InfoViz 是一款将数据表转换成可视图形的工具。
译注 7：GraphViz 是一款显示结构化信息的图形工具。

先尝试后决策

Try Before Choosing

埃里克·多伦伯格（Erik Doernenburg）



创建一个应用需要作出许多决策。有些决策涉及挑选框架和函数库，而另一些则需要选择特定的设计模式。不管哪种决策都需要架构师拿主意。平庸的架构师可能会收集手边的信息，斟酌酝酿一番，然后从象牙塔里颁布解决方案让开发人员实现。无疑，还有比这更好的方法。

玛丽·波彭代克（Mary Poppendieck）和汤姆·波彭代克（Tom Poppendieck）夫妇俩（译注 1）在著作中描述了一种制订决策的技巧。他们主张尽量推迟决定的时间，最后即便团队不作决策，决策也会自己呈现——等待不易逆转的结果。这是种审慎的技巧，因为越晚作出决策，可利用的信息就越多。但是在很多情况下，“更多的信息”并不等于“充足的信息”，而且完美的决策只可能出现在事后。那么架构师应该如何使用这种技巧呢？

架构师应该持续关注那些马上要制订的决策。假设团队中有多位开发人员，并且代码所有权属于整个开发队伍，架构师可以在决策时间点到来之前，要求几个开发人员商量出解决方案，尝试一段时间。当决策时间点临近时，召开会议比较不同解决方案的优点和弊端。通常由于做过尝试，这时大家对问题的最佳

译注 1：玛丽和汤姆夫妇是第 14 届 Jolt 大奖获奖图书《Lean Software Development: An Agile Toolkit》的作者，这本书的中文版《敏捷软件开发工具——精益开发方法》由清华大学出版社出版。

的解决方案已经有了共识。架构师只须组织协调制订决策的过程即可，不必自己作出决策。

这个方法既适合决定简单的问题，也适用于复杂的问题。既可以帮助团队决定是否使用 Spring 框架提供的 Hibernates 模板，也可以用来决定挑选哪个 JavaScript 框架。但很显然，问题的复杂度会影响制订决策的时间。

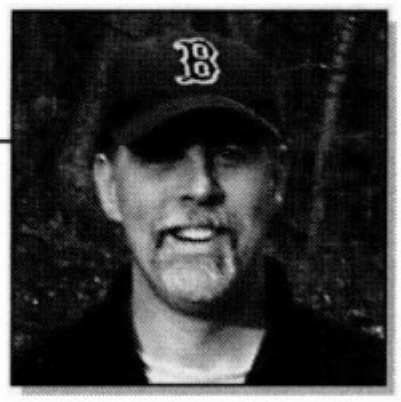
对同一个问题尝试两种或两种以上的解决方案，比直接决策然后动手实现的工作量要大。但是，如果事后发现仓促决定的方案不合适，架构师将陷入进退两难的困境：无论是放弃目前的方案还是继续开发，都会造成误工和损失。更糟糕的情况是没人发现方案不合适，在这种情况下，甚至察觉不到损失。总之，尝试多种解决方案可能是代价最低的选择。

作者简介见第 57 页。

掌握业务领域知识

Understand the Business Domain

马克·理查兹（Mark Richards）



高水平的软件架构师不仅要懂技术，还要掌握问题空间（problem space）对应的业务领域知识。缺乏业务领域知识的架构师不能顺利地理解业务问题，无法把握业务目标和业务需求，也就难以设计有效的架构来满足需求。

架构师的角色任务在于理解业务问题、业务目标、业务需求，并设计技术架构来满足它们。掌握业务领域知识将有助于架构师选择合适的架构模式，更好地制订针对未来的扩展计划，适应不断变化的产业趋势。举例来说，有些业务领域（如保险行业）本身的特点很适合采用面向服务的架构方法，而其他业务领域（如金融市场）更适合采用基于工作流的架构方法。掌握领域知识，可以帮助我们挑选出最能满足客户具体需求的架构模式。

掌握具体领域的行业趋势同样有利于设计有效的架构。例如在保险行业，市场对“按需投保”类汽车险种（“on-demand” auto insurance）（译注 1）的需求正在不断增长，投保人可以按实际驾车时间交纳保费。假设你周一早上把车停在机场，乘飞机前往工作地点，周五才飞回来，然后驾车回家，这类保险会非常

译注 1：这种保险业务又称为 Pay-as-you-drive car insurance，它根据实际驾车时间来计算须交纳的保险费。它既节省了投保人的开支，又鼓励大众使用更环保的方式出行，因而受到越来越多车主的青睐。

适合你。把握这些行业趋势，你甚至能赶在公司把它们纳入商业模型之前，就在架构中做好准备。

理解具体的业务目标也有助于你设计有用的架构。例如可以问问自己，客户的业务目标是否要考虑由于合并和收购引起业务量急剧增长的情况。不同的答案将影响架构的类型。如果要考虑这种情况，那么架构就需要包括若干抽象层，用来降低合并业务组件的难度。如果客户计划通过大规模的在线产品展示来增加市场份额，那么持续高可用性（high availability）就是关键的质量属性。作为架构师，你要始终理解公司的业务目标，并确保架构支持这些目标。

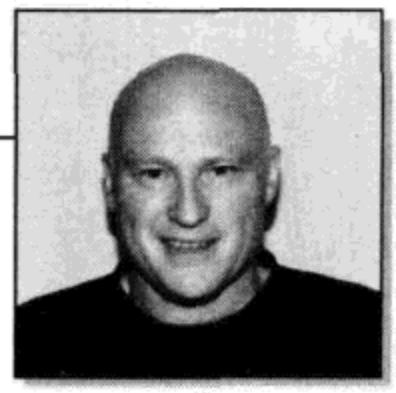
我认识的那些成功架构师不仅拥有丰富的、通过实践积累的技术知识，同时也对特定的业务领域知识了如指掌。他们能够自如地运用企业高管（C-level executives）和用户熟悉的行业术语与他们沟通。这反过来又增强了软件架构师对自己工作的信心。掌握业务领域知识有助于软件架构师更好地理解要解决的难题、有争议的问题、业务目标，以及数据和流程。这些都是设计有效企业架构的关键因素。

作者简介见第 9 页。

程序设计是一种设计

Programming Is an Act of Design

埃纳尔·兰德雷（Einar Landre）



发明面向对象程序设计方法和 Simula 程序设计语言的克利斯滕·尼高（Kristen Nygaard）博士曾说过，程序设计是学习的过程。将程序设计——或者更准确地说，软件开发——看作发现和学习的过程，而不是生产和建造的过程。这种观点从根本上促进了软件实践方法的发展。传统的生产和建造理念不适合运用于软件开发。30 年来，软件行业的思想先驱不断就此发表观点，并且留下了许多文字记录。例如小弗雷德里克·布鲁克斯（Fredric Brooks, Jr.）（译注 1）曾在 1987 年的《国防科学委员会军用软件任务组报告》(Report of the Defense Science Board Task Force on Military Software）中指出，以文档驱动的、照着说明依样画葫芦（specify-then-build）的方法是引发许多软件问题的罪魁祸首。

如果要发展实践方法，软件行业应该到哪里取经呢？不妨看看汽车、医药和半导体这些生产大众产品的精密行业。

来看看汽车行业的做法。计划推出新车型时，首先要决定设计概念和原型。这相当于设计架构的过程。以宝马 X6 为例，这款新车型结合了多功能运动车（SUV）和双门轿跑车（coupe）的特性，被宝马公司称为全能轿跑车。X6 面市之前，宝马公司已经在车型设计和制造设计上投入了数千个小时和数百万的资金。宝马公司接到顾客的订单后，会启动某条流水线，根据顾客要求生产定制的 X6。

译注 1：弗雷德里克·布鲁克斯因担任 IBM 的大型机 System/360 及其操作系统 OS/360 的开发项目经理，被称为“S/360 之父”，著有《人月神话》(The Mythical Man-Month: Essay on Software Engineering)、《计算机体系结构：概念与发展》(Computer Architecture: Concept and Evolution）等。

我们从汽车的生产过程中可以获得什么启示呢？新车的生产过程包含两个阶段，第一个是创新的设计过程，包括建立投产必备的流水线；第二个阶段才是根据顾客的要求生产汽车。从许多方面看，软件行业也是如此，只是划分的界线不同。

杰克·里夫斯（Jack Reeves）在《什么是软件设计？》（What is software design?）这篇文章里提到，在软件工程里，与传统行业的设计文档具有相同地位的只有源代码。软件的生产则是自动化的，由编译器、构建工具和测试代码共同完成。

如果把编写代码看成设计行为，而不是生产行为，我们就能采用一些已经被证明有效的管理方式。这些方法过去用于管理具有不可预测性的创新工作，比如研发新车、新药、新的电脑游戏。我所指的是敏捷的产品管理方法和精益生产方法，比如 SCRUM，它们关注如何为客户实现最大的投资收益。

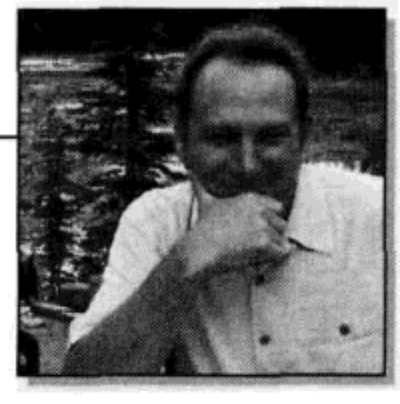
如果软件行业希望从这些方法中获益，我们必须记住，程序设计属于设计范畴，而不是生产范畴。

作者简介见第 13 页。

让开发人员自己做主

Give Developers Autonomy

菲利普·尼尔森（Philip Nelson）



多数架构师都是从开发人员干起的。架构师在决定如何构建系统的工作中肩负着新的责任，也拥有更大的权力。刚上任的架构师会发现很难沿用以往的工作方式开展新工作，总是迫切地觉得自己还需要大量的练习才能胜任管理开发人员的工作，让大家实现设计。应该给予团队成员足够的自主权，让他们发挥自己的创意和能力，这对你和团队来说都是件好事。

以前作为开发人员，你很少有机会坐下来仔细观察整个系统是怎样组合在一起的，而作为架构师，这是你工作的重点。开发人员忙于编写类、方法、测试代码，使用接口和数据库，你则要确保所有这些东西良好地协调运作。要善于倾听各种抱怨并设法加以改进。编写测试代码的人遇到了麻烦？请改进接口降低依赖性。你知道哪里需要抽象，哪里不需要吗？请理清问题所属的领域。你知道构建系统的正确顺序吗？请制定项目计划。开发人员在使用你设计的 API 时，是不是总犯同样的错误？请修改设计方便同事理解。大家真的理解你的设计吗？请通过沟通来表达清楚。你知道哪里需要实现可伸缩性，哪里不需要吗？请和客户一道学习他们的业务模型。

如果你想出色地完成架构师的工作，是不可能有空闲去干预开发人员的。虽然你应该密切注意团队是否在按计划实现系统，但是没必要站在背后监视大家。当你发现同事遇到麻烦时，可以主动给出建议。但更可取的做法是创造良好的氛围，让大家主动向你征求意见。这件事如果做好了，不但能确保架构成功实现，而且能让团队成员把智慧和创意发挥到极限。

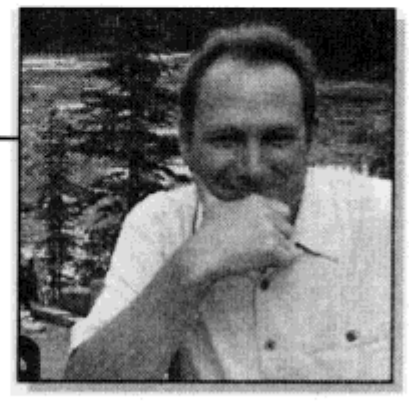
作者简介：

菲利普·尼尔森是个技术通才，他先后从事过硬件、网络、系统管理方面的工作，最后转向软件开发和架构设计，他觉得这些工作一个比一个有趣。他处理过不同领域的软件问题，包括：交通、金融、制造业、市场营销，以及一些与基础设施相关的问题。

时间改变一切

Time Changes Everything

菲利普·尼尔森（Philip Nelson）



我喜欢随着时间的流逝观察哪些事物会被淘汰，哪些能保存下来。自诩聪明的人们曾经怀着激情提出过那么多的模式、框架、范式转换（paradigm changes）和算法，哪一个不是声称可以一劳永逸地解决所有已知问题？结果都不过是昙花一现。为什么会这样？历史在向我们昭示什么？

选择值得投入精力的工作

Pick a Worthy Chalienge

这个要求对软件架构师来说有些棘手。问题和任务总是给定的，我们没有选择的余地，不是吗？并非这么简单。首先，我们常常误认为自己无法左右交给我们的任务。其实我们可以，只不过这要求我们从舒适的技术领域里迈出来。如果不选择做正确的事情，会为将来埋下隐患。随着时间流逝，我们勤勤恳恳、兢兢业业地完成了给定的任务，最后却发现做得毫无意义：因为实现的不是真正的需求，白费了工夫。漂亮的解决方案搭配正确的任务，才能经受时间的考验。

简单原则

Simple Rules

人们反复念叨 KISS（keep it simple，stupid）原则，把它挂在嘴边，却没有身体

力行地去遵守。大家不遵守是因为觉得没必要。我们自认为聪明，可以应付复杂的问题，而且轻易就能找到各种借口替自己的行为辩护：例如为了增加设计的灵活性，为了让设计更优雅，或者为了更符合我们对美感的要求。我们相信自己能够预见未来。但随着时间流逝，当我们回顾一年以前甚至更早的项目时，几乎都会惊诧自己当初的做法。如果有机会再做一次，我们会以更简单的方法来完成。这就是时间的作用，时间让我们的行为变得可笑。越早醒悟越好，别再自以为是，用时间的放大镜仔细研究简单原则的真正涵义吧。

别跟以前的工作过不去

Be Happy with That Old Stuff

架构师们喜欢寻找“真正的解决途径”：可以提供明确答案、具有预见性的方法论或思想流派。但似乎总也找不到。问题在于，你现在看重的设计思路，可能两三年后就会被自己否定，更不要说十年。当你回顾过去的工作时，永远会觉得以前的设计和自己的期望有差距。学着接受以前的工作吧，克制自己回过头去修改的冲动。那个解决方案适合吗？它能不能解决需求？把这些问题当作今后的工作标准，你的心情会好很多。

作者简介见第 65 页。

设立软件架构专业为时尚早

"Software Architect" Has Only Lowercase a's; Deal with It

巴里·霍金斯（Barry Hawkins）



软件开发行业近来出现了一种令人不安的趋势：有人急于将软件架构的设计工作专业化，与传统的建筑学专业平起平坐。起因似乎是某些软件架构师不满足于同事和老板的肯定，希望自己的成就获得更正式的认可。要知道建筑学专业直到 19 世纪末才确立，在此之前建筑这个行当已经存在了几千年。两相比较，这些软件架构师未免太心急了。

设计软件架构是一门手艺（craft），从业者无疑要通过实践和训练才能在这个领域里获得成功。但是，软件开发仍然处于相对初级的尝试阶段，我们对它的了解还相当不够，不足以将其专业化。虽然软件开发诞生的时间不长，但是它的成果作为一种工具，已经获得了极高的评价。所以事业有成的架构师（还有那些装作事业有成的架构师）的薪酬已经达到主流专业人士的水平，与医生、会计师和律师相当。

从事软件开发可以享受丰厚的薪酬，因为它是富有创意，具有探索性质的工作。许多重大的历史性突破得益于我们的工作成果，有些成果甚至惠及全人类。但进入软件行业的门坎主要是个人的实力和机遇；而那些发展成熟的专业，对从业者有着严格的学习和实习要求。从这一点来看，我们的行业还很稚嫩。扪心自问，我们还缺乏充分的理由将其专业化，让软件架构师加入律师、医生和建筑师的行列为时尚早。

在设立软件架构专业之前，我们还有很长的路要走。

作者简介：

巴里·霍金斯是土生土长的亚特兰大人，拥有 13 年的软件开发经验，从独立开发者到团队主管，再到敏捷教练和顾问，他担任过各种工作。目前他的主要工作是担任敏捷软件开发的教练和顾问，同时研究领域驱动设计。

控制项目规模

Scope Is the Enemy of Success

大卫·奎克（Dave Quick）



规模指的是软件项目的大小。完成项目需要多长时间、多少人工和资源？要实现哪些功能？质量上有什么要求？难度如何？风险有多大？要遵守哪些约束条件？这些问题的答案界定了项目的规模。软件架构师都喜欢挑战复杂的大型项目。由于回报非常诱人，甚至有人搞“形象工程”，故意夸大规模。殊不知规模越大，项目失败的可能性越大，这一点常让人始料不及。规模扩大一倍，失败的可能性往往会增加十倍。

为什么呢？请看两条前人的经验：

- 凭直觉判断，只要花两倍的时间或资源，就能完成两倍的工作任务。但是历史（注 1）告诉我们直觉不可靠，这里不存在简单的线性关系。例如，比较四个人组成的团队和两个人组成的团队，前者花在沟通上的时间肯定超过后者的两倍。
- 估算与准确的科学计算相差甚远，所以产品特性实现起来常常比预期的要困难。

当然，有些项目正是因为具有一定的规模和复杂性，才有开发的价值。好比一款文本编辑器，必须具备文字输入的功能，否则它压根就不是文本编辑器。那么，有哪些策略可以帮助我们缩小和控制项目的规模呢？

- 抓住真正的需求。软件项目的交付目标表现为一组需求，需求定义了软件的功能和功能的质量。不能为客户创造价值的需求应该遭到质疑。如果实现一项需求不能为公司带来收益，就应该放弃。

注 1：请参考《The Mythical Man-Month: Essays on Software Engineering》，作者弗雷德里克·布鲁克斯（Frederick Brooks），Addison-Wesley Professional 出版；本书中文版《人月神话》由清华大学出版社出版。

- 分而治之。寻找机会将大项目分解成独立的小项目。比起由相互依赖的子系统组成的大项目，几个相互独立的小项目更容易管理。
- 设置优先级。业务环境瞬息万变，大型项目在完工之前，需求会改变多次。虽然有些需求会随业务变化甚至被取消，但是关键需求通常会维持不变。理清需求的优先级，优先实现最关键的需求。
- 尽快交付。看到演示产品之前，多数客户都不知道自己想要什么。有一幅漫画流传很广，画的是根据客户的描述搭造秋千的过程。漫画展现了不同的人对客户需求的不同理解，这些人把需求想得太复杂，几乎与秋千不沾边。最后一格漫画配的文字是："实际需求原来如此"，画面上是一条悬挂着在半空中的废旧轮胎。让客户试用演示产品，没准会发现更简单的解决方案。首先实现最重要的需求，尽快获得客户的反馈，越快越好。

敏捷方法的倡导者提倡开发"最简单有用的东西"（the simplest thing that could possibly work）（注 2）。越复杂的架构越难成功实现。缩小项目规模通常会降低架构的复杂性，这是架构师提高成功机率最有效的途径。

作者简介见第 51 页。

注 2：请参考《eXtreme Programming eXplained: Embrace Change》，作者肯特·贝克（Kent Beck），Addison-Wesley Professional 出版；本书中文版《解析极限编程—拥抱变化》由人民邮电出版社出版。

架构师不是演员，是管家

Value Stewardship Over Showmanship

巴里·霍金斯（Barry Hawkins）



架构师接手新项目，都渴望证明自己的价值，这是人之常情。公司除了要求架构师具备无可争议的技术领导能力，也希望他怀揣证明自己能力的迫切愿望。遗憾的是，有些架构师误解了“证明自己价值”的含义，以为是炫耀技术才华，甚至是刁难开发团队。

炫耀和作秀是市场营销的重要手段，可以吸引顾客，却与指挥开发项目背道而驰。架构师必须扎实地掌握技术和业务领域的知识，以严谨的领导风格赢得团队的尊重。

架构师的职责和管家类似，承担着管理他人资产的责任。所以架构师应该尽可能为客户的利益着想，不能存有私心。

软件架构要满足不同领域的客户需求，而这些领域的专业知识通常是架构师所不具备的。为了成功完成项目，架构师必须计算可用的时间和人力，综合考虑成本和复杂性等因素，设计出折中的解决方案。这里的时间和人力是公司托付给架构师管理的资源，作为“管家”的架构师绝不能暗藏私心。卖弄时髦的软件框架和流行的技术词汇，只会把系统变得更复杂，给公司造成损失。架构师的工作和投资代理很相似，记住，客户之所以允许别人动用自己的资金，是为了获得满意的投资回报。

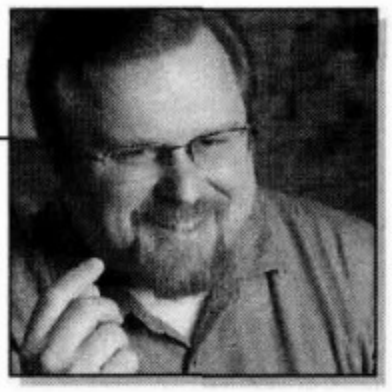
架构师不是演员，是管家。别忘了，你花的是客户的钱。

作者简介见第 69 页。

软件架构的道德责任

Software Architecture Has Ethical Consequences

迈克尔·尼加德（Michael Nygard）



软件世界的道德范畴边界并不清晰。尽管有些行为无疑是不道德的，比如侵犯他人的公民权利、窃取他人身份信息、使用流氓软件等，但还有些行为的道德意义却不容易察觉。流行的软件产品可以影响成千上万的用户，影响可以是正面的，也可能是负面的——好软件节省时间，糟糕的则反之——即使这种影响不过几分钟。

软件架构师的每项决策（例如设置必填项和规定流程），都限制了用户可以做什么，不能做什么。这比制定法律容易得多。即使用户嫌这些“必填项”和“规定流程”麻烦，也找不到法院受理他们的诉讼。

我们可以从倍增效应的角度来看待软件的影响。回忆一下最近网络病毒爆发的情形，或者科幻大片上映的盛况。那些发表在媒体上的分析和评论，每次都用耸人听闻的数字指责它们（病毒和大片）让大家无法安心工作，造成巨大的社会损失。这些想必你早有耳闻。我举这个例子不是为了指责媒体的无知和夸大事实的作风，而是为了说明倍增效应对我们的影响。

假设架构师要实现一项新功能。采用简单的设计一天能完成，采用复杂的设计需要大概一周时间，你会怎么选择？如果简单的设计要求用户填写四个必填项，而复杂的设计可以巧妙地处理用户不完整的输入数据，你会怎么选择？

虽然设计必填项从表面上看并无不妥之处，但实际上是架构师在强迫用户接受自己的意图。必填项迫使用户准备更多的信息，否则无法开始工作。用户不得不把资料逐个记录在记事贴上，直到所有的数据收集齐全。这个过程中常常会丢失数据，耽搁工作，让人非常沮丧。

打个比方，店铺外要挂一块新招牌，把它安装在约一人高的地方最省事，免得搭梯子和脚手架，反正不会把路堵死，路人可以弯腰通过，或者绕道。可你是否想过，你节省的时间是以浪费他人时间为代价的。从长远来看，路人为此浪费的时间，将远远超过你省下的时间。

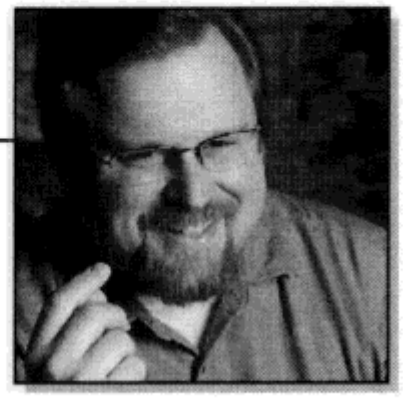
损人利己是不道德的行为，哪怕程度很轻。成功的软件产品会影响数以百万计的用户，而用户不得不无奈地接受架构师的每项决策，所以永远不要忽略了架构师对用户的影响，这是我们勇于承担责任、减轻用户负担的动力。

作者简介见第 17 页。

摩天大厦不可伸缩

Skyscrapers Aren't Scalable

迈克尔·尼加德（Michael Nygard）



大家经常把软件工程比喻成建造摩天大厦、水坝和公路，这种比喻在某些方面确实有道理。

土木工程不仅是设计建筑对象这么简单，真正的难题在于规划整个施工过程，确保建筑物拔地而起，包括从奠基到竣工的所有工作。在竣工之前，建筑工人必须各施所长，让建筑框架一直竖立着。其中有些经验很值得我们借鉴，尤其是对于布署大型集成化软件系统（包括所有的企业应用和 Web 应用）。如果把软件工程比作土木工程，那么传统的“大爆炸”式软件部署方式（“big bang” deployments）（译注 1）就好比把备齐的建筑材料一股脑儿扔上天，指望它们瞬间拼成大厦一样可笑。

相反，无论是开发新项目，还是替换已有的系统，都应该逐个部署系统组件。这样做有两个优点。

首先，隐藏在代码中的技术风险是部署软件时无法回避的问题，每个组件都有可能出问题，一次只部署一个组件，可以将风险分散到各个时间段，每次砌好“一块砖”。

其次，这种方法迫使我们设计清晰的组件间接口。在旧系统上部署新组件，相当于将新组件回归集成（reverse-integrating）（译注 2）到旧系统里，如果新系统部署成功，那么所有新组件就同时适用于两套系统：旧系统和替换后的新系统。组件是否可重用必须通过实践检验，采用分段式部署（piecewise deployment）方法可以有效检验组件的可重用性。实践证明，这种方法还能提高系统的一致性，降低系统的耦合度。

译注 1：“大爆炸式”部署是指用新系统彻底替换旧系统，新旧系统之间不存在过渡阶段的布署方式。对复杂的系统来说，这种一刀切式的做法风险比较高。

译注 2：回归集成是版本控制系统中的术语，指将分支版本（尤其是其中修改后内容，例如新开发的功能）合并到主版本库。

但是，还有些土木工程的经验是不能借鉴的，尤其是在建筑工程中屡试不爽的“瀑布式”（waterfall）施工方法。要知道，摩天大厦的选址和建筑高度在设计时已经确定，事后加高楼层不但代价高昂，而且非常危险。设计定稿后，就不能再改变大厦的高度，更不能挪位置。毕竟，摩天大厦不需要可伸缩性。

与拓宽马路相比，增加软件的功能要简单得多。这不是软件开发过程的瑕疵，恰好是软件的优势。应用软件只要具备了用户要求的关键功能就可以发布，不必等到十全十美才上市。事实上，产品越早发布，公司的净收益值就越高。

表面上看，“尽早发布”（early release）的做法和“递增式部署”（incremental deployment）似乎相互矛盾，但两者其实可以相辅相成。尽早发布单独的组件，意味着每个组件都可以独立地迭代。这迫使你提前解决诸如划分协议版本、维持部署过程中的持续可用性等棘手的问题。

尽早部署独立的组件既可增加商业利润，又可以改善架构品质，这样实用的技巧实在不多。

作者简介见第 17 页。

混合开发的时代已经来临

Heterogeneity Wins

爱德华·加森（Edward Garson）



随着计算机技术的“自然进化”，架构师用来构建软件系统的工具发生了重大的变化。这种变化再次激起了人们对混合编程（polyglot programming）的兴趣。混合编程是指在同一套软件系统中同时采用多种核心编程语言。

混合编程不是新的概念，以前就出现过，比如大家曾经熟悉的一种架构：前端采用 Visual Basic 开发客户端，后端采用 C++的 COM 对象开发应用。当时这种架构充分发挥了两种语言的优势。

那么，是什么样的变化重新激发了大家对混合编程的兴趣呢？

得益于技术标准的发展，以及不断增加的带宽和计算资源，现在可以采用基于文本的协议（text-based protocols）了。在分布式系统领域，晦涩难懂的二进制协议已经不再是提高效率的前提条件。基于 XML/SOAP 的 Web 服务首先采用了基于文本的远程协同工作方式（remote interoperability），随后 Restful 架构风格、Atom 和 XMPP 进一步促进了基于文本协议的发展。

这些新技术以特定格式的文本作为载体，便于所有人编写和理解，为混合开发（heterogeneous development）提供了前所未有的可能性。混合开发方式允许为每项任务挑选最合适的工具，基于文本的协同工作方式则敲开了通往混合开发的大门。

架构师可以把若干个强大的开发工具组合起来使用，以往的标准是挑选合适的编程语言，现在则演变成挑选合适的编程范式。编程语言支持的范式各有不同，有些是面向对象的，有些是函数式的，有些适合并发编程。对于具体的领域问

题，有些范式可以完美地解决，有些则显得力不从心。但没关系，现在可以很容易地把看似不搭界的工具“混搭”（mash up）起来使用，形成巧妙的解决方案。

混合开发的效果已经开始显现，并且正以组合性增长（combinatorial increase）的态势扩展软件系统的架构拓扑图。它一方面反映了软件系统特有的多元性，同时也指出了新的发展方向。

虽然选择多了并不总是件好事，但至少好过以往软件架构非此即彼的窘境。软件行业正面临着严峻的挑战（注 1），现有的开发平台还不足以应付新出现的问题（注 2），我们必须发掘更多的协同工作方式才能化险为夷。

新的技术变革正逐步瓦解我们以往积累的技术成果，架构师将面临更大的挑战。我们应该拥抱这种变化，跳出原有的思维模式，充分挖掘软件的多元化特性。混合开发的时代已经来临。

作者简介：

自打在 Apple II 电脑上学习用 LOGO 语言编程起，爱德华·加森就迷上了软件技术。他目前供职于 Zuhlke Engineering，这是一家总部设在瑞士的技术公司。他在公司的敏捷实践中心担任架构师。

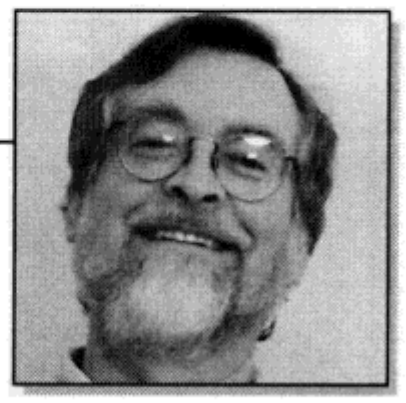
注 1：软件开发在即将到来的多核处理器时代很可能遭遇前所未有的挑战。

注 2：参见赫伯·萨特（Herb Sutter）的文章《The Free Lunch is Over》（*http://www.gotw.ca/publications/concurrency-ddj.htm*）。

性能至上

It's All About Performance

克雷格·罗素（Craig Russell）



假设有这样一款轿车，空间宽敞、乘坐舒适，不但省油，而且价格低廉，98%的配件都是可循环利用的，你会动心吗？当然了，这样的车人人想要。如果最高车速只有 10 公里/小时（6 英里/小时），你还会动心吗？这个例子可以说明性能和其他指标一样重要。

有些设计师把性能问题放在最后考虑。他们觉得与人比起来，计算机的工作速度已经够快了，用户肯定可以接受系统的工作速度，既使现在系统速度不够快，还可以指望摩尔定律发挥作用。但是硬件速度不是系统的一切。

我们通常把系统响应用户输入的时间作为衡量性能的标准。其实系统设计师要考虑的性能问题不止于此，它们还包括系统分析师和程序员实现设计的效率、系统的人机交互性能，以及非交互组件的性能等。

生产率通常用来描述构建系统的效率，也属于性能范畴，其重要性在于直接影响项目的成本和进度。进度落后或预算超支可都是大麻烦。使用现成的组件，合理利用已有的工具，可以显著提高系统的构建效率，提前创造收益。

系统的人机交互性能直接关系到用户是否愿意掏钱。人机交互性能与很多设计因素相关，其中最为人熟知的是响应时间。除了响应时间，影响交互性能的因

素还包括界面是否直观，操作（使用）步骤是否简单。

合格的说明书除了注明系统每秒钟的响应次数，还要测量典型的任务时间。任务时间是完成指定领域任务的时间与人机交互时间之和。任务时间在响应时间的基础上增加了用户思考的时间和用户输入数据的时间，这些都是不受系统控制的。考虑这些时间，可以促使我们完善人机界面的设计。通过关注信息的展现方式，分析必要的交互步骤，可以挖掘更人性化的设计思路，提高交互性能。

非交互性组件的性能同样影响着系统的表现。比方说，设置在夜间运行的批处理任务，如果拖到早上还没完成，势必影响系统白天的正常工作。此外，灾难恢复组件的性能也至关重要，如果局部系统遭到破坏导致业务中断，它直接决定了恢复正常工作状态所需的时间。

在考虑系统的实现方法和运维策略时，架构师和设计师应该密切关注系统的性能表现。

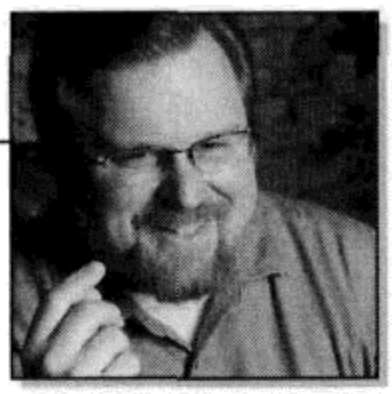
作者简介：

克雷格·罗素是 Sun 公司的高级工程师，主要研究对象持久化和分布式系统的架构设计。

留意架构图里的空白区域

Engineer in the White Spaces

迈克尔·尼加德（Michael Nygard）



软件系统由相互依赖的程序组成，我们把装配这些程序的方法及程序之间的关系称为架构。绘制架构图时，一般用简单的矩形表示这些程序，用箭头表示程序之间的关系。

假设某个箭头代表了“使用 HTTP 协议，发送 SOAP-XML 格式的同步请求/响应消息”（Synchronous request/reply using SOAP-XML over HTTP）。由于架构图的空间有限，写不下这么多内容，所以通常用简单的注释来表示。从技术角度出发，可以简写成“XML over HTTP”，如果从业务角度出发，有可能写成“查询库存单元”（SKU Lookup）。

不同的程序看似通过箭头直接联系，其实不然。矩形之间的空白区域“充满”了各种软件和“硬件”，比如：

- 网卡
- 网络交换机
- 防火墙
- 网络入侵检测系统（IDS）和入侵防御系统（IPS）
- 消息队列或消息中介
- XML 转换引擎
- FTP 服务器
- “临时上传区”分配表
- 城域同步光纤环路
- 多协议标记交换（MPLS）网关
- 网络干线（Trunk lines）
- 海洋
- 可能破坏海底光缆的拖网捕鱼船

通常 A 程序和 B 程序之间至少隔着 4～5 台主机。这些主机上运行着各种软件，包括分组交换、流量分析、路由计算、威胁分析，等等。这些“隐藏在空白区域中的因素”是设计架构时必须考虑的。

我见过一个箭头的注释是“提供满意的服务（Fulfillment）”。箭头一端是客户公司的服务器，另一端是一台远程服务器。这个箭头代表的不是一个简单的接口，它承担着为客户提供满意服务的重任。它由一系列复杂的事件组成，包括将消息代理（message broker）接收到的消息转换成文件，周期性地通过 FTP 协议传输等，超过 20 个步骤。

应该理解每个箭头包含的静态信息和动态信息。除了“SOAP-XML over HTTP”这样的注释，还应该知道箭头包含的动态信息：“利用单个 HTTP 请求发送查询，利用单个 HTTP 响应接收回复。每秒钟最多能发送 100 次请求。响应时间控制在 250 毫秒以内，成功率应该达到 99.999%”。

除此以外，还有更多的细节要考虑：

- 如果会话发起方的请求频率太高怎么处理？接收方可以忽略请求吗？还是应该“礼貌地”拒绝，或者尽可能地响应？
- 如果响应时间超过了 250 毫秒，会话发起方该怎么处理？是重新发送请求，继续等待，还是默认接收方出现了故障，跳过这个请求执行后续步骤？
- 如果会话发起方发送请求用的是 1.0 版的协议，收到的回复却是 1.1 版的，该怎么处理？如果收到的不是 XML 格式的文件，而是 HTML 格式的文件，甚至是 MP3 格式的文件，又该怎么处理？
- 万一会话双方中的一方暂时没有响应，该怎么处理？

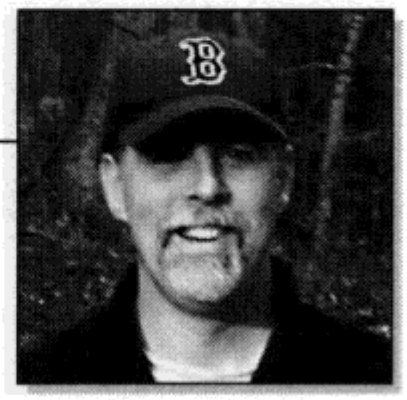
只有想清楚了，才能顺利地解决空白区域里的问题。

作者简介见第 17 页。

学习软件专业的行话

Talk the Talk

马克·理查兹（Mark Richards）



每个专业都有行话，同行之间讲行话方便交流。律师经常使用人身保护权（habeas corpus）、预先审查（voir dire）之类的术语；木匠口头离不开平接接头（butt joint）、搭接接头（lap joint）、乳胶（flux）；软件架构师则经常提到 ROA、两步视图、层超类型（layer supertype）。但你清楚这些行话的意义吗?

为了提高效率，软件行业有一套可行的办法，供不同平台的架构师相互交流，其中包括使用架构模式和设计模式。架构师想要提高工作水平，必须掌握基本的架构模式和设计模式，学会识别不同的模式，并借助它们和同行及开发人员进行交流。

架构和设计模式可以分成四大类：企业架构模式、应用架构模式、集成模式、设计模式。每类模式对应着整体架构的不同层次和范围，例如企业架构模式解决高层的架构问题，而设计模式则研究架构中每个组件的构造方法。

企业架构模式定义架构的全局框架结构。常用的模式包括事件驱动型架构（EDA）、面向服务的架构（SOA）、面向资源的架构（ROA），还有流水线型（pipeline）架构。

应用架构模式指出了全局架构下的子系统及局部应用的设计方法。常用的模式包括大家熟悉的 J2EE 设计模式（包括会话外观（Session Façade）、传输对象等），以及马丁·福勒（Martin Fowler）在著作《Patterns of Enterprise Application Architecture》（Addison-Wesley Professional 出版）（译注 1）中提到的应用架构模式。

译注 1：这本书的中文版《企业应用架构模式》由机械工业出版社出版。

集成模式研究怎样在系统的组件、应用和子系统之间传递信息，共享功能，它是设计和交流重要手段。集成模式包括文件共享、远程方法调用，以及不同的消息传递模式。各种集成模式的详细介绍请参考：*http://www.enterpriseintegrationpatterns.com/eaipatterns.html*。

“四人帮”（译注 2）（Gang of Four）在《Design Patterns: Elements of Reusable Object-Oriented Software》（Addison-Wesley Professional 出版）（译注 3）中描述的基本设计模式，是每位架构师必须掌握的知识。虽然这些模式讨论是较底层的设计内容，离架构师的工作比较远，但是它们是架构师和开发人员相互交流的标准行话，可以提高沟通的效率。

除了以上四类模式外，架构师还应该了解并提防各种反模式（anti-pattern）。反模式指的是影响软件开发结果的常见错误，这个词是安德鲁·凯尼格（Andrew Koenig）发明的。常见的反模式包括需求分析麻痹症（Analysis Paralysis）、委员会设计（Design By Committee）、蘑菇管理（Mushroom Management）、死亡征途（Death March）等。学习反模式可以避免许多容易疏忽的问题。各种常见的反模式请参考：*http://en.wikipedia.org/wiki/Anti-patterns*。

用清晰、简洁、高效的方式与同行进行沟通，是软件架构师应该具备的能力。前人总结了许多现成的模式，架构师只有学习和理解它们，才能同时提高工作水平和沟通能力。

作者简介见第 9 页。

译注 2：“四人帮”是软件行业对《Design Patterns: Elements of Reusable Object-Oriented Software》四位作者的习惯称呼。

译注 3：这本书的中文版《设计模式：可复用面向对象软件的基础》由机械工业出版社出版。

具体情境决定一切

Context Is King

爱德华·加森（Edward Garson）



分享设计架构的理念让我觉得很滑稽，因为我认为压根儿就不存在设计理念。如果我是对的，就无从下笔了，好在我是个矛盾的人，愿意冒险做些太阳打西边出来的事儿。

毕竟，没有理念本身就是一种理念。

对我来说最重要的设计经验是：具体情境（context）决定一切，根据它设计尽量简单的解决方案。换句话说，架构决策只有在情境需要时，才能牺牲尽量简单的原则。

我说的情境不仅指业务目标这样高层的、直接的因素，还包括其他外界因素，比如各种新兴的技术和创新的思想。称职的架构师应该多关注发展迅速的新技术。

高品质的架构是由什么构成的？架构是在具体情境下作出的一系列决策，用以实现一组通常相互制约（竞争）的需求。由于需求常常相互制约，所以设计架构的关键不是贡献新内容，而是忽略那些不必要的需求。设计架构的过程其实就是作出明智决策的过程（产品则体现了架构师的设计意图）。

具体情境影响架构的先例很多，其中不乏生动的故事，比如大家津津乐道的挑选 M1 艾布拉姆斯主战坦克（M1 Abrams tank）车载数据库的例子（注 1）。（当

注 1：虽然制造坦克的目的可疑，但不可否认设计难度非常大。

然，挑选数据库通常不是设计架构的重点，举这个例子只是为了说明应用情境的重要性。）

M1 坦克的设计团队评估了许多数据库。坦克必须在起伏不平的地形上高速行驶追踪目标，导航系统和觅标系统的数据吞吐量很大。设计团队发现大部分数据库可以满足最严格的要求，可是坦克主炮射击产生的巨大电磁脉冲会彻底破坏车载软件系统，包括数据库。在现代战场上，没有软件的坦克只能像瞎子一样乱窜。在这种情况下必须迅速恢复数据库，当时 InterBase 数据库的表现最出色（注 2），所以被选为 M1 坦克的车载数据库。

我觉得论坛讨论组上那些激烈争论技术优劣的人，不过是茶余饭后寻消遣。他们争论的并非悬殊的技术差异，不过是些细微的不同之处，更何况脱离了具体的应用情境，孤立地比较技术的优劣是毫无意义的事。

别让团队成员被各种设计理念束缚住，鼓励大家具体情况具体分析，努力找出最简单的解决方案。

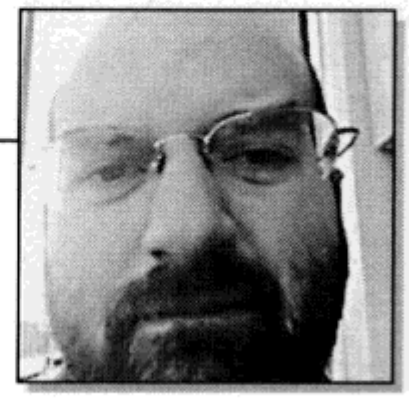
作者简介见第 79 页。

注 2：InterBase 数据库的磁盘写操作可以保证数据库与磁盘始终处于一致的状态，这是它能迅速完成灾难恢复的原因。

侏儒、精灵、巫师和国王

Dwarves, Elves, Wizards, and Kings

埃文·考夫斯基（Evan Cofsky）



在尼尔·史蒂芬森（Neal Stephenson）的小说《CRYPTONOMICON》（EOS 出版社）里，主人公兰迪·沃特豪斯（Randy Waterhouse）把自己遇到的人分成三类。侏儒（dwarf）最勤劳，他们住在黑暗的洞穴里，孤独坚韧地制作精致的工具，并发挥惊人的力量，移山填海。他们以精湛的手艺著称。精灵（elf）最有风度和修养，他们擅长制作新奇的魔法物品。虽然他们天赋很高，却不知道其他种族把他们的作品看成异类。巫师（wizard）与这两个种族不同，他们拥有无限的魔力。他们比精灵更了解魔法的秘密和力量，能够施法创造奇迹。此外，还有第四种人，沃特豪斯虽然提及却没有明确归类，那就是国王。国王拥有团结所有种族的能力。

软件架构师好比国王，应该熟悉各种人的性格特点，招聘不同性格的人加入自己的团队。由一帮性格相同的人设计的架构只能吸引同样性格的人加入团队，即使你拥有一群最棒的“侏儒”（“精灵”或“巫师”），也会由于视野不够宽阔，只能用单一的方法解决问题。

英明的国王知道怎样用目标来激励不同的种族，率领大家并肩作战完成任务。如果没有目标，团队看不到希望，就会内讧；如果大家的特点相同，只能解决一类问题，也无法完成最后的解决方案。

架构师安排任务时，应该时刻考虑所有开发人员的性格特点。从这个角度来看，架构是一个指南，为不同性格的团队成员安排合适的任务，让大家在工作过程中相互学习。如果大家有机会充分磨合、相互适应，就能轻松化解各种难题。

作者简介：

埃文·考夫斯基是位软件工程师，业余时喜欢作曲、骑自行车。他上大学时学的是作曲和计算机科学，现在仍然在持续学习。目前他在 Virgin Charter 网站担任高级软件工程师，是网站的常驻 Python 专家，和一群优秀的、多才多艺的同事共事。

向建筑师学习

Learn from Architects of Buildings

基思·布雷思韦特（Keith Braithwaite）



> 建筑是社会性的表演，是上演人类历史的剧院。
>
> ——斯皮罗·科斯托夫（Spiro Kostof）（译注 1）

有多少软件架构师还把自己的工作看成单纯的技术工作？难道我们不曾周旋于各种利益集团之间，充当和解人、中间人，甚至仲裁者的角色？有多少人对待工作时，还是一副清高的知识分子态度，不愿正视这份工作必须和人打交道？

> 要想成为伟大的建筑师，优雅丰富的心灵远比聪明才智重要。
>
> ——弗兰克·劳埃德·赖特（Frank Lloyd Wright）（译注 2）

哪种架构师更容易脱颖而出？是那些天资聪明、对技术细节烂熟于心的人，还是那些宽容、文雅、高尚的人？你更愿意与谁共事？

> 医生可以缝合手术创口掩盖医疗事故，建筑师只能建议业主栽种爬山虎。
>
> ——弗兰克·劳埃德·赖特

维护自己设计的系统远比“修剪爬山虎”麻烦。你有勇气删掉有缺陷的代码吗？还是假装没看见？赖特认为锤子和榔头才是建筑师最好的朋友。最近你推翻过自己的设计吗？

译注 1：斯皮罗·科斯托夫（1936~1991 年），出生于土耳其，著名建筑历史学家，他认为所有建筑都有着深刻的人文指向，著有《城市的形成》和《城市的组合》等。

译注 2：弗兰克·劳埃德·赖特（1867~1959 年），美国著名建筑师、作家，以设计“有机建筑”著称，他的最著名的作品包括：东京帝国饭店、流水别墅、纽约古根海姆博物馆等。著有《自传》、《消失的城市》、《有机建筑》等。

> 建筑师自诩上帝的助手，甚至觊觎上帝的宝座。
>
> ——卡伦·莫耶（Karen Moyer）

这句话里的“上帝”应该换成“客户”。

> 盖房子和别的手艺一样，所有努力都是为了结果，为了完成令人满意的工作。合格的建筑应该符合以下三个条件：实用、坚固、令人愉悦。
>
> ——亨利·沃顿（Henry Watton）

最后一次见到让你赞不绝口的架构是什么时候？你是否立志要让自己的程序给别人带去愉悦？

> 建筑师首先应该是伟大的雕塑家，或者伟大的画家，否则他不过是个建筑工人。
>
> ——约翰·拉斯金（John Ruskin）（译注 3）

你的架构是否蕴涵适当的艺术的成分？用组件搭建的系统，有没有借鉴绘画的造型和质感？有没有从雕塑的姿势和平衡中汲取灵感？是否考虑了适当留白的重要性？

下面这句毋须再加评论，对架构师最常犯的毛病来说，它无疑是一剂良方。

> 这似乎是个诡异的悖论，但也是最重要的真理：天底下没有完美的建筑。
>
> ——约翰·拉斯金（John Ruskin）

作者简介见第 21 页。

译注 3：约翰·拉斯金（1819~1900 年）英国著名的作家、艺术家、艺术评论家，拉斐尔前派的成员。其著作和哲学观念对艺术与手工艺的发展有着深远的影响。著有《现代画家》系列、《拉斐尔前派》等。

避免重复

Fight Repetition

尼克拉斯·尼尔森（Niclas Nilsson）



你的开发人员在重复无须思考的工作吗？代码里反复出现某些相似的片段？某些代码是复制粘贴后略加修改而成的？如果出现这些情况，说明团队工作效率不高。别不信，罪魁祸首很可能是你。

听我解释之前，先来看看两条公认的软件开发的真理：

- 复制是魔鬼。
- 重复性的工作拖累开发进度。

作为架构师，你的工作直接影响团队的开发风格。你最了解系统的全貌，没准还写了一个端对端的系统垂直切片（vertical slice）作为示范——一个反复使用过的示例。如果开发人员复制示例代码里的内容——无论是几行代码、XML文件，还是类——说明这些部分还可以简化，甚至全部抽取出来（abstracted away）。复制的部分通常不属于领域逻辑（domain logic），而是承担底层工作的基础性代码。因此，架构师一定要警惕示例可能会产生的影响。这些代码和配置将成为成百上千个系统切片的模板，应该做到简洁、意图明确，只包含无法抽取掉的、与领域问题相关的内容。架构师应该对那些可能重复的内容保持高度警惕，因为你的每行代码都会（不可思议地）被复制。

不相信自己的系统会出现这样的情况？检查一下配置文件吧，假设把它用于其他系统切片，哪些地方需要修改？哪些地方可以保留？再瞧瞧典型的业务层方法是不是存在与其他方法重复的部分，比如事务处理、登录、身份验证、审核等？还有数据访问层，除了实体名和字段名以外，有没有其他内容重复出现？然后加大搜索范围，系统里是不是总有两三行代码反复在一起出现，虽然每次操作的对象不同，其实功能是一样的？以上这些都是典型的重复。开发人员阅读代码时，迟早会忽略掉重复出现的代码，因为他们会逐渐找出规律，知道哪些才是有价值的代码。但是，就算开发人员找出了规律，重复的代码还是会影响大家的开发效率。这样的代码只适合交给计算机执行，不适合阅读和交流。

消灭重复的内容是你的责任，为此，应该重新研究框架，创造更完善的抽象机制，请专门制作工具的程序员(toolsmith)帮你完成切面框架(aspect framework)，或者使用代码生成器。要想消灭重复内容必须有人采取行动。

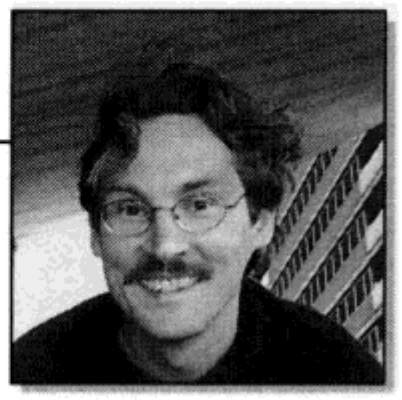
这个人就是你。

作者简介见第 31 页。

欢迎来到现实世界

Welcome to the Real World

格雷戈尔·侯珀（Gregor Hohpe）



工程师偏爱精确，整天与 1 和 0 打交道的软件工程师更是如此。我们习惯了非此即彼的世界，凡事不是 1 就是 0，不是真就是假，不是是就是否。在外键约束、原子事务和校验和的保证下，一切都是清楚的、确定的。

可惜现实世界不是二进制的。顾客有可能撤销确认过的订单，支票可能跳票，信件可能丢失，付款时间可能延误，许下承诺还可能失信，数据记录时不时就会出错。用户偏爱提供更多功能的“压缩”（shallow）界面，他们讨厌繁琐的、一步接一步的交互步骤。对程序员来说，这种一维的过程看起来更符合逻辑，也更容易实现。可是别忘了，真正决定程序流程的不是调用堆栈（call stack），而是用户需求。

这些已经够糟了，可分布式系统又带来了新的不一致性。服务有可能失效，状态可能在毫无征兆的情况下改变，事务处理可能得不到保证。应用程序运行在成千上万台机器上，出错是不可避免的，只是迟与早的问题。分布式系统不但是松耦合、异步、并发的，而且不遵守传统的事务语义（transaction semantics），这些都是程序员的噩梦。

计算机科学家设想的完美世界正在崩溃，我们该怎么办？克服所有困难的步骤都一样，首先要接受现实。向令人怀念的调用堆栈架构告别吧，忘掉那些程序

员决定程序流程的日子，准备好应付随时出现的乱序事件，不断根据具体情境调整策略。用异步的、并发的请求代替一个接一个的方法调用。设计应用时，借助事件驱动的过程链（event-driven process chain）模型或状态模型控制无序的状况，通过调整、重发，甚至试探的办法纠正错误。

你也许不曾料到事情这么复杂吧。还好，类似的问题也一直在现实生活中上演：信件延误、承诺后毁约、通信相互干扰、收款账户混淆——这样的例子举不胜举。我们的对策是重发信件、撤销定单、通知付款人钱已到账，不必再理催款单，等等。所以别再抱怨现实世界带来了麻烦，不妨从中寻找解决问题的灵感。比如向星巴克学习，看看他们是怎么改良两阶段提交（two-phase commit）模式（注 1）的。欢迎来到现实世界。

作者简介：

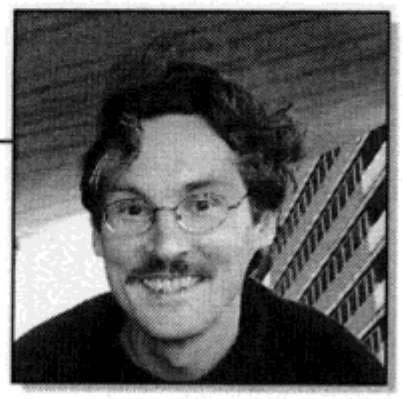
格雷戈尔·侯珀是 Google 的软件架构师。他是业界公认的异步消息架构及 SOA 方面的权威，曾参与编写《Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions》（中文版《企业集成模式：设计、构建及部署消息传递解决方案》由中国电力出版社出版），并定期在技术会议上发表演讲。

注 1：请参考：*http://www.eaipatterns.com/ramblings/18_starbucks.html*。

仔细观察，别试图控制一切

Don't Control, but Observe

格雷戈尔·侯珀（Gregor Hohpe）



我们已经进入分布式、松耦合系统的时代。构建松耦合的系统多少有些麻烦，为什么大家要自讨苦吃呢？因为我们希望系统足够灵活（flexible），别因为一点小小的变动就支离破碎。分布式应用只有一小部分受本地控制，其余部分则以分布式服务或第三方软件包的形式存在，由其他部门或软件厂商控制，所以灵活性是分布式系统的重要指标。

看来设计可以随着时间不断灵活变化的系统是个好主意。但这首先意味着系统会不断变化，就像人们常说的“日新月异”，而且记录系统文档将是个大麻烦。一般写下来的文档信息都不是最新的，这点相信大家都有体会。而要描述不断变化的系统，情况只会更糟。此外，灵活的系统意味着更复杂的架构，因而更难描绘出“全局视图”。比方说，如果系统组件是通过可配置的逻辑信道通信的，就得查看所有的信道配置才能了解通信双方的具体情况。在分布式系统里，消息发错信道要比编译错误严重得多，因为每条消息都与用户的利益密切相关。

妄想掌控一切的架构师只能设计出紧耦合的、脆弱的解决方案，这一套已经行不通了。但是放任自流也不行，那只会导致系统混乱。我们必须启用必要的辅助机制。使用仪表飞行（instrument flight）（译注 1）必须有仪表设备，不是吗？

译注 1：指飞行员根据飞机仪表指示的状态架驶飞机。仪表飞行技术是在复杂天气条件下飞行的主要技术。

有哪些“仪表”可供架构师选用呢？选择其实很多。现在的编程语言已经开始支持反射技术，几乎所有的运行时平台都提供运行时测量标准（runtime metrics）。随着系统的配置越来越灵活，当前的系统配置包含了更多信息，为了便于理解，必须从中提取模型。如果你搞清楚了哪个组件负责向逻辑信道发送消息，哪个组件负责接收消息，就应该把组件间的通信关系用图表模型记录下来。这件事可以每隔几分钟或几小时做一次，在系统的变化过程中，记录下最新的、准确的系统模型。不妨把这个过程看成“反向 MDA”（Reverse MDA）（注 1）。与模型驱动架构不同，你先构建出灵活的架构，然后再从实际的系统状态中提取模型。

多数情况下，模型很容易画出来，得到系统的全局视图也不困难，但是千万别用巨型广告板把系统中的所有类和依赖关系详细画下来。这张图作为艺术品也许不错，作为软件模型则太不实用。埃里克·多伦伯格（Erik Doernenburg）已经介绍了正确的做法（译注 2），应该采用“一千英尺高”的视图。选择合适的抽象层次能为你提供有效的信息，也方便你用基本的验证规则检查模型，比如检查依赖图中是不是存在循环依赖（circular dependencies），发送到逻辑信道的消息是不是都有接收方负责接收。

撒手不管是很危险的状态，对系统架构来说也是一样。对 21 世纪的架构师来说，正确的做法应该是：仔细观察，提取模型，然后检查验证。

作者简介见第 95 页。

注 1： MDA 是模型驱动架构（model-driven architecture）的简称。

译注 2：请参见本书第 28 篇《使用“一千英尺高”的视图》一文。

架构师好比两面神

Janus the Architect

大卫·巴特利（David Bartlett）



在罗马神话里，两面神（Janus）是司守门户和万物始末之神。他有两张面孔，凝视着两个相反的方向，这个形象经常出现在硬币上和电影里。两面神象征着生命的流逝变迁，生老病死、婚丧嫁娶、岁月蹉跎。

两面神兼顾前与后、过去与未来的能力应当受到所有架构师的推崇。架构师要在不同的对象之间架起桥梁，比如梦想与现实、过去的成功与未来的方向、业务（管理）目标与开发限制，等等。在完成项目的过程中，由于不同矛盾因素的介入，架构师常常要跨越各种鸿沟。例如既要让系统易于访问，又要保证系统安全；既要让设计符合当前的业务流程，又要体现管理层对未来发展规划的考虑。所以架构师必须具备两面神的能力，融合不同的思想和观念，兼顾不同的设想和目标，才能开发出皆大欢喜的产品。

两面神不光是长了两张脸，他其实长了两个头，比常人多出一对耳朵和一双眼睛，拥有额外的警觉能力。优秀的 IT 架构师也应该善于倾听和观察。例如了解公司基本开支的用途，可以更好地推测管理层的发展规划；评估开发人员的设计能力和技术能力，可以有针对性地安排培训，挑选合适的结对编程队员，更好地完成项目；熟悉哪些开源软件可以和商业软件搭配使用，可以有效地节约项目开支、缩短开发时间。优秀的架构师应该眼观六路、耳听八方，收集各种信息提高开发效率。

把架构师比作两面神，并不是说架构师应该像神一样挑剔，要求架构完美无缺。称职的架构师应该勇于接受新观念，敢于尝试新的设计思路和工具，促进项目、团队，甚至整个行业的发展；他不会浪费大把的时间参加管理层会议，或者妄想独自编写所有的代码；他应该采纳好点子，营造活跃的思考氛围。只有思想开放的架构师才能平衡各种矛盾因素，顺利地完成项目。架构师都希望带领团队完成项目获得成功，但是优秀的架构师设计的系统不但易于维护、方便扩展，还能经受时间流逝、业务发展，以及技术升级的考验；优秀的架构师善于倾听、观察、斟酌，重视重构自己的设计、过程、方法，从而确保项目的质量；他们付出的所有努力，都是为了产品能够经受岁月的洗礼。

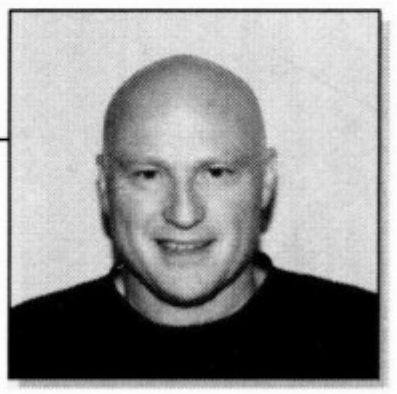
每位架构师都应该追求这种境界。但是知易行难，我们应该以两面神为榜样，工作上严格把关，综合考虑新情况与老经验，在成熟技术的基础上不断创新，既满足当前的业务需求，又兼顾未来的发展规划。

作者简介见第 41 页。

架构师当聚焦于边界和接口

Architects' Focus Is on the Boundaries and Interfaces

埃纳尔·兰德雷（Einar Landre）



自纳尔逊勋爵（Lord Nelson）在 1805 年的特拉法尔加（Trafalgar）海战中摧毁了法西联军舰队以来，“分而治之（divide and conquer）”已经成为处理复杂难题的神咒。表达相同的意思但更为人所知的另外一个术语，则是“关注分离（separation of concern）”。为了分离关注点，人们发明了封装（encopsulation）的办法，从封装又引出了边界（boundary）和接口（interface）的概念。

从架构师的角度看，困难的所在，是要找到设置边界的自然之处（natural place），并定义出构建可工作系统（a working system）所需的合适接口。大型的企业系统，其自然边界稀少及多个领域之间互有纠缠，做到这点尤其困难。在此情况下，古老的智慧诸如“低耦合，高内聚（Minimize coupling, maximize cohesion）”和“避免横向切分需要高度信息交换的区域（Do not slice through regions where high rates of information exchange are required）”提供了一些指导方针，但它们并没有提及如何以一种简易的方式与利益相关者（stakeholder）就问题和潜在解决方案（potential solutions）进行沟通。

埃里克·埃文斯（Eric Evans）在他的著作《领域驱动设计（Domain-Driven Design）》（Addison-Wesley Professional）中描述的“有界情境（bounded context）”和“情境地图（context mapping）”的概念，对此颇有助益。有界情境是用以唯一定义一个模型或概念的区域，通常以一朵云或一个气泡来表示，同时还赋予一个在当前领域中能够定义其角色和职责的描述性名称。举例而言，在航运系统中，可能会包括诸如“货运操作”、“货运调度”和“港口运送”情境。在其他领域中，也有相应的适合名称。

识别出这些有界情境并在白板上绘制出来之后，就到了开始绘制出各个情境之间关系的时候了。这些关系可能会揭示出在组织、功能或技术上的依赖。这项工作的成果，便是一个情境地图（context map），它包含了一系列有界情境及它们之间的接口。

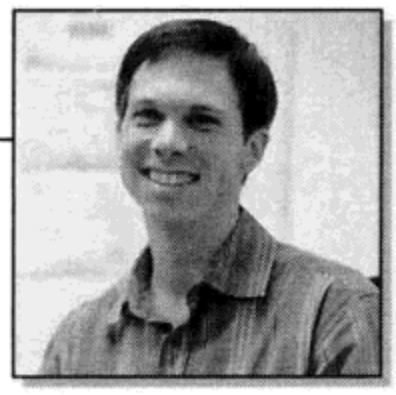
这样的一个情境地图，为架构师提供了一种强大的工具，使得他们可以聚焦于“哪些应该归在一起，而哪些应该分开”，从而使他们能够以一种可顺畅沟通的方式，实施明智的分而治之。他们可以十分容易地采用该技术来记录和分析系统的当前的状况（as-is situation），并从该点出发，继续指导对系统的重新设计，以获得一个具有低耦合、高内聚和接口定义良好（well-defined interfaces）等特征的更好的系统。

作者简介见第 13 页。

助力开发团队

Empower Developers

蒂莫西·海伊（Timothy High）



诸事说易行难，而软件架构师“对未来之事特能吹嘘”这点已是“臭名昭著”。为了避免你的言辞最终变成蒸腾的热气（它通常是制造雾件（译注 1）的关键元素），你需要一个优秀的开发团队。作为一名架构师，你的角色通常是要去施加约束，但是，你也有机会成为推动者（enabler）。你应该在职责范围之内，尽量助力开发团队。

要确保开发人员拥有他们所需的工具。工具不应该强行规定，而应当仔细选择，确保它们是开发人员处理手头工作的正确工具。应当尽可能地自动化那些重复和无须动脑筋（mindless）的工作。同时确保开发人员拥有一流的机器用于工作，拥有足够的网络带宽，可以访问到开展工作所必需的软件、数据和信息，这些也是非常值得的投资。

要确保开发人员拥有所需的技能，确保他们能够获得必需的培训。在书籍上进行投资，并促进在技术方面开展积极的讨论。开发人员在工作期间必须要动手和实践，但同时也要有活跃的学术研讨。如果有专项预算，请送团队去参加技术专题讲座和研讨会；如果没有，那么让他们参与到技术主题的邮件列表中去，同时在你所在城市里寻找类似的免费活动。

同时，也要尽可能地参与到开发人员的选拔过程中。挖掘那些热衷于学习，那些有那么一点“亮点（spark）”表明他们能真正去钻研技术（同时也要注意确

译注 1：雾件（vaporware），指某项产品或技术，在面世之前通过炒作而备受关注和期待，可实际上却一拖再拖，只闻其声不见其影，在市面上始终无法买到，像雾一样，看得到而摸不着。

保他们能够默契地融入团队……）的开发人员。一个充斥平庸之辈（duds）的团队很难产生爆炸性的巨响（a big bang）。

只要不违背软件设计的总体目标，就让开发人员自己做出决策。但是，不仅为了确保质量，也为了能够对开发人员进行深度授权，我们要在总数上设置限制。为了一致性（consistency），也为了减少麻烦和无关紧要的决定（它们不是开发人员要解决的根本性问题的构成部分），我们要创建一些标准。创建一个清晰的路线图（roadmap），向开发人员说明应当如何放置源代码文件，如何对之命名，何时应当创建新文件等诸如此类的问题。这可以节省他们的时间。

最后，保护好开发人员，不要让他们卷入到不那么重要的工作中。过多的文牍工作和办公室杂务增加了总开销，降低了开发人员工作的有效性。你一般不会是管理人员，但你可以对周边的软件开发过程施加影响。无论使用的是何种过程，要确保它们的设计目的是消除障碍，而不是增加障碍。

作者简介：

蒂莫西·海伊（Timothy High）是一个在 web、多层客户机-服务器架构（multitiered client-server）和应用集成技术方面拥有超过 15 年经验的软件架构师。他目前作为一名软件架构师，供职于 Sakonnet Technologies，它在能源交易和风险管理（Energy Trading and Risk Management，ETRM）领域处于领先地位。

记录决策理由

Record Your Rationale

蒂莫西·海伊（Timothy High）



在软件开发社区，对于文档尤其是关于软件自身设计的文档的价值，争论颇多。分歧一般集中于两处，一处是“详细的前期设计（big upfront design）”的有效价值，另一处则是使设计文档和不断变化的代码库保持同步的难易程度。

记录软件架构决策理由的文档，长期有用，又无须为之付出过多维护精力，具有很高的投资回报价值。正如马克·理查兹（Mark Richards）在《取舍的艺术》一篇（译注 1）中所说的，定义软件架构，就是要在质量属性、成本、时间以及其他各种因素之间，做出正确的权衡。此份文档应能向你自己、经理人员、开发人员及软件的其他利益相关者，清楚阐明选择某种解决方案，而非另外一种的原因，包括其中做出的权衡。有没有打着减少硬件和许可费用的幌子，牺牲了系统的水平伸缩性（horizontal scalability）？虽然缩短了数据交换的整体响应时间，但数据加密是否足够安全？

根据项目的不同，可以灵活选择合适的文档格式来记录架构决策的方方面面，格式可以是文本、维基（wiki）或博客（blog）形式的速记备忘录，也可以使用更为正式的模板。无论使用何种形式和格式，此文档都应回答以下基本问题：“我们做了什么决策？”“为什么这样决策？”有一个稍次要的问题，经常有人会问到，因而也要记录下来：“我们还考虑过哪些解决方案？为什么没有采用？”（事实上，有人经常会这样问：“为什么我这个方案不行？”）。此文档应该放在容易查找，可被搜索的地方，以备不时之需。

译注 1：参见本书第 22 篇《取舍的艺术》。

这类文档迟早会派上用场，比方说：

- 可以作为和开发人员进行沟通的工具，说明应遵循的重要架构原则。
- 当开发人员对架构背后的逻辑提出质疑时，使团队成员能够“就事论事”，甚至能够避免一场“兵变（mutiny）（译注 2）”（如果事实表明你的决策站不住脚，便应虚心接受批评）。
- 向经理和利益相关者说明这样构建软件的确切原因（比如，采用某种较为昂贵的硬件或软件的必要性等）。
- 要把项目移交给下任架构师时（你有多少次在接手软件时很疑惑原来的设计者究竟为什么要采用那种做法？）。

然而，从这种实践中可获得的最重要好处是：

- 它逼着你明确说出理由，有助于确保基础（foundation）是扎实稳固的（参见下一条文章《挑战假设尤其是你自己的》）。
- 如果相关条件发生变化，需要对决策重新评估，它可以作为一个起点。

创建此种文档，只需在相关主题的会议或讨论中随手做些速记备忘即可。无论选择什么样的格式，这类文档都物超所值。

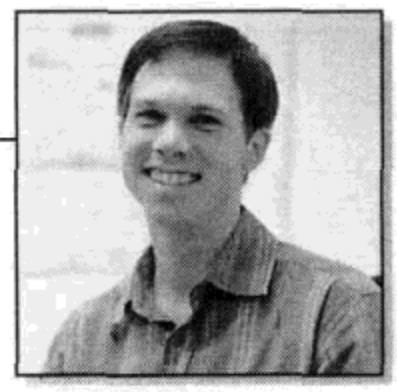
作者简介见第 103 页。

译注 2：指架构师和团队成员间不是进行“就事论事”的沟通，而是进行带有人身攻击性质的争论，最终破坏团队的健康。

挑战假设，尤其是你自己的

Challenge Assumptions—Especially Your Own

蒂莫西·海伊（Timothy High）



韦森“延期判决”法则（译注 1）（wethern's law of suspended judgment）以诙谐口吻如是说：“臆断是事情搞砸的根源（Assumption is the mother of all screw-ups）。”另一种更为流行的说法是“不要假设（assume）——它会让你我出丑（make an 'ass' of 'U' and 'me'）（译注 2）。”但是，如果面对的是可能导致数千甚至数百万美元损失的假设，你的心情就不会那么轻松了。

软件架构的最佳实践表明，应该记录下每个决策背后的理由（译注 3），当这一决策包含权衡（性能 vs.可维护性，成本 vs.上市时间等）时尤须如此。在更为正式的方法中，记录下每个决策的上下文是很常见的做法，这些上下文包含了促成最终决策的各项“因素”。这些因素,可能是功能性或非功能性需求，但也可能只是决策者认为重要的“事实”（或道听途说...），如技术约束、现有技能、政治环境因素等。

这种做法颇有价值，因为列出这些因素有助于标明（highlight）架构师所持的假设，这些假设会影响到软件设计中的重要决策。很多时候，这些假设往往是基于“历史原因”、主观判断、开发人员的视野、因循守旧（FUDs），甚至“走廊里听来的一些小道消息”：

- “开源软件不可靠。”
- “位图索引（Bitmap index）带来的麻烦比好处多。”
- “客户绝对不会忍受一张网页需要花 5 秒钟才加载完毕。”
- “只要不是主要厂商销售的产品，首席信息官（CIO）都会拒批。”

译注 1：韦森“延期判决”法则是软件开发社区中流行的诸多诙谐法则之一。更多可见 *http://www.scs.uiuc.edu/suslick/laws.html*。《持续集成》一书中也曾引用过此法则。

译注 2：注意 assume 这个单词的拼写，由“ass”+“u”（you）+“me”组成。

译注 3：参见上篇《记录决策理由》。

确保这些假设清楚明确，对后继者和未来重新评估来说，非常重要。但是更重要的是，一定要拿出相关的经验证据（或者获得参与者的确认），仔细验证过这个假设之后，才可以做出最终决策。如果客户为了关键报表愿意接受 5 秒的等待时间，而你提供的却是相反的信息，那该如何是好？如何确认某个开源项目确实是不可靠的？对于“在数据中使用位图索引”，是否通过应用程序的事务（transactions）和查询（query）对之进行测试？

特别提醒，不要忽略“相关（relevant）”这个词。老版本软件中正确的一些东西，今天可能已经不再成立了。位图索引的性能，在 Oracle 某个版本中和在另一个版本中，可能并不相同。某个旧版本的类库中可能存有的安全漏洞，现在没准已经被修复了。可靠的老牌软件厂商在财务上可能已是苟延残喘。技术格局每天都在变化，人也是一样。谁知道呢？也许你的 CIO 私底下已经变成为 Linux 的狂热拥趸了。

事实（fact）和假设（assumption）是构建软件的两大支柱。务必确保软件的基石坚实可靠。

作者简介见第 103 页。

分享知识和经验

Share Your Knowledge and Experiences

保罗·W·霍默（Paul. W. Homer）



从所有得失成败的经验中，我们可以学到很多东西。在像软件开发这般年轻的行业中，为了持续发展（sustain progress），传播经验和知识至关重要。每个团队在自己的小世界小角落里所学到的东西，可能会在全球产生影响力。

实际上，要成功开发项目，仅凭软件开发的基础知识，即那些在理论上绝对正确的知识，是远远不够的。其他不足，我们依靠猜测（guess）、直觉判断，甚至随机挑选来弥补。就此而言，任何主要开发项目的成败经验都是经验性的证据。我们期望能够通过不断的交流，将之整体反馈回行业中去。

在个人层面，我们都在努力成长，不断认识应当如何构建更大型的系统，在职业生涯中也遭遇越来越大的挑战，因此期望之前的经验能提供指导。亲临现场当然不错，但要从经验中获得最大启示，就还要经常进行理性总结（rationalize）。为了达到这个目标，最好、最容易的方法，就是尝试向他人解释。

讨论有助于发现不足。只有能非常容易地做出解释，才表明你真正理解了。只有不断解释和讨论，才能把经验凝聚成知识。

另外要补充一点，从过往经验得出的推论，并不完全适用于所有情况。我们也许没有想象的那么成功，或如期望的那么聪明。当现实检验表明，从未置疑过的东西忽然被证实是缺乏根据，不正确或压根儿就从没正确过时，确实令人颇为灰心，无法接受；承认错误确实不易。

归根结底，我们都是凡人，因此所知的一切不可能都是正确的，我们的每个想法并不都是合理的。只有勇于接受不足，才有不断改进的可能。从失败中能够学到更多，是颠扑不破的真理。如果思想和信仰经不起辩论的考验，那么，现在发现，总比以后重新建立要好。

我们真诚希望能够通过分享知识和经验，帮助业界持续发展；我们也认识到，这有助于更好地理解和修正已知的知识和经验。软件五花八门，利用一切机会分享我们知道的、我们认为我们知道的，以及我们已经发现的，其重要性不言而喻。如果能帮助周围的人不断改善，他们也会帮助我们发挥出全部的潜力。

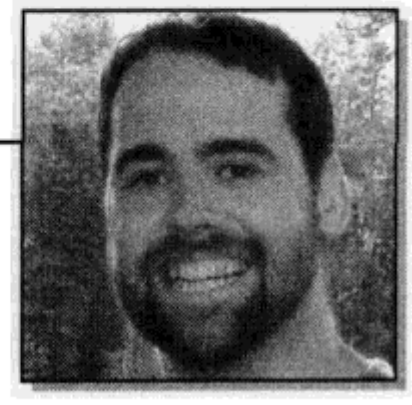
作者简介：

保罗·W·霍默（Paul W. Homer）是名软件开发人员、作家，偶尔摆弄摄影，自数十年前被卷入软件开发行业至今，他一直在不断努力构建日益复杂的系统。

模式病

Pattern Pathology

查德·拉·瓦因（Chad La Vigne）



对于软件架构师来说，设计模式是极有价值的可用工具之一。使用模式，能够创建出更易沟通、更易理解的通用解决方案。模式与良好设计直接相关。这一事实，让“在项目中摆出大量模式展示非凡的架构功力”显得非常诱人。如果发现自己试图把最喜欢的模式硬套在不适用的问题空间（problem space）上，那么你也许是“模式病（pattern pathology）”患者。

许多项目遭遇过这种情况。可以设想，在这些项目中，富有独创性的（译注 1）架构师（the original architect）翻完模式宝典的最后一页之后，搓着双手说：“现在，该首先使用哪一个模式呢！？”。这种思维方式，有点像开发人员开始编写某个类时想“嗯，应该对哪个类进行扩展呢？”。对于减轻必要的复杂性（necessary complexity）而言，设计模式是非常优秀的工具，但和其他工具一样，它们也可能被滥用。如果设计模式成了俗话所说的到处敲钉的锤子（译注 2）时，问题就来了。要小心，不要让你对模式的喜欢，变成了迷恋（infatuation），进而引入超出实际所需的过于复杂的解决方案。

译注 1：此处作者是以诙谐的口吻说出的一个反语。

译注 2：拿着锤子，到处都是钉子，是软件工程中的著名俚语和典故，指代“以一种方法对待全部问题”的思维模式和问题解决方法。

在项目中硬塞进不必要的模式，是过度工程（over-engineering）。设计模式不是魔法，在解决方案中使用它并不能确保获得好的设计。它们是对常见问题的可重用解决方案。人们记录发现的模式，避免后来人重新发明车轮（wheel）（译注 3）。当这些方案能够解决的问题出现时，我们能够识别出来，并恰当地应用设计模式，这才是我们的任务。不要让一展设计模式功力的欲望，遮蔽了务实的真知（pragmatic vision）。应当保持对系统的洞察力，提供切实有效的商业解决方案，使用模式解决适用的问题才是最重要的。

作者简介：

查德·拉·瓦因（Chad La Vigne）是一位解决方案架构师和技术专家，供职于巴尔的摩郡的 TEKSystems 公司。他主要在明尼阿波利斯（Minneapolis）地区工作，使用企业级 Java 技术来设计和实现解决方案。

译注 3：“不要重新发明轮子”是软件工程中的另一句著名俚语，指对于前人已经获得稳定解决方案的问题，就不要浪费时间重新求解，而要复用。

不要滥用架构隐喻

Don't Stretch the Architecture Metaphors

戴维·英格（David Ing）



架构师喜欢使用隐喻（metaphor）。对那些通常比较抽象、复杂和变化移动的目标，隐喻提供了很好的具体媒介。无论是与其他队员沟通，还是与最终用户讨论架构全局，找到有形实物作为正要构建的东西的隐喻，都是十分诱人的。

开始这很有效，使用一种共同语言，也能让大家都感觉到正确的方向，不断演化前进。随着时间推移，隐喻不断发展成长起来，栩栩如生。人们对隐喻感觉良好——我们正在不断前进！

常见的情况是，对于架构来说，之前的那些隐喻现在变得危险了。滥用架构隐喻经常会出现问题让架构师不知所措，比如：

- 业务领域的客户开始越来越喜欢系统隐喻，这时系统还在构想中，在这种情况下，所有各方共享的是最乐观的可能解读（happiest possible interpretation），但其中并没有包括任何必要的约束。

举例而言："我们正在构建一个运输系统，就像在一系列停靠点之间移动的运输船一样。"

你想的是横渡太平洋的集装箱轮船。而我想的，其实只是在游泳池中的单桨划艇。

- 开发团队开始认为隐喻比实际业务问题更重要。由于团队耽溺（fondness）于隐喻，你不得不开始修正那些古怪的决策。

举例而言："我们说过，它就像一个文件柜，当然要按字母顺序显示给用户。我知道它是个 6 维的、没有容量限制并且内置时钟的文件柜，但我们现在已经做好图标了，因此它必须就是一个文件柜……。"

- 所交付的系统包含了许多遗留名称，从早已老旧过时、有待重新鉴定的隐喻，到多次重构和重复挖掘的概念。

例如："为什么账单工厂对象（Billing Factory object）要为划艇系统创建一个港口通道（Port channel）？当然它应该为中心总线（Hub Bus）返回一个石榴视图（Pomegranate view）？你说什么，你是新来的？"

请记住，不要耽溺于系统隐喻之中，只将之用于探索性的沟通，不要反让它拖了后腿。

作者简介：

戴维·英格（David Ing）是名软件架构师和技术专家，在不列颠哥伦比亚省的温哥华生活和工作。他原来居住在英国，为摆脱多雨天气跨洋过海，尽管现在感觉深受不诚实的旅游文学之骗。

跟进时尚潮流发展，他现在为一家 Web2.0 公司——Taglocity 工作。他的工作时间分为两部分，一部分时间用于努力使电子邮件系统"更好用"，另一部分时间用于弄清 Web 2.0 究竟意味着什么。

关注应用程序的支持和维护

Focus on Application Support and Maintenance

门西蒂西·卡斯珀（Mncedisi Kasper）



应用程序的支持和维护永远都不应该是事后才考虑的事情。由于应用程序超过80%的生命周期都是在维护上，在设计时就应该多多关注支持（support）和维护（maintenance）的问题。忽略这一点，你将会惊恐万分地注视着寄予厚望的应用程序停止工作，宛如失控的野兽，跌入恐怖的死亡深渊，成为你架构师生涯中无法抹去的败绩。

在设计应用程序时，大多数架构师主要是站在开发人员的角度思考，他们手上有 IDE 和调试器。当问题出现时，高度熟练的软件工程师会进行调试来发现错误。由于架构师大多出身于开发人员，而非系统管理员，他们很容易会按这种思路来考虑问题。不幸的是，开发人员和支持人员拥有的技能不同，就像开发/测试环境和生产环境有截然不同的目的一样。

下面列举了系统管理员会遇到的一些问题：

- 系统管理员不能重新提交请求消息来重现问题。在生产环境中，也不能对“线上”数据库重发资金交易来查看何处出了问题。
- 一旦应用程序进入了生产环境，修复错误的压力来自于客户和管理人员，不是项目经理和测试团队，而愤怒的总裁将意味着更大的威胁。
- 一旦投入使用，就没有调试器可用了。
- 一旦投入使用，部署工作就需要进行计划安排和协调的。无法把生产环境中的应用程序停几分钟，来测试错误修复的情况。
- 生产环境中的日志记录级别要比开发环境中的低很多。

如果支持规划存在此种缺陷，就会出现下面这些症状：

- 大多数问题都需要开发人员的参与。
- 开发团队和支持团队之间的关系很疏离沉闷，开发人员认为支持团队不懂技术。
- 支持团队讨厌新的应用程序。
- 架构师和开发团队在生产环境上花了很多时间。
- 经常把重启应用程序当作一种解决问题的办法。
- 管理员总是在救火，他们永远都没时间把系统调整到合适状态。

为了确保应用程序脱离开发人员之手后能成功运行，应该做到：

- 理解开发人员和支持人员确实具有不同的技能。
- 在项目中尽可能早地引入支持负责人。
- 将支持负责人作为团队的核心成员之一。
- 让支持负责人参与规划应用程序的支持。

如此设计，则技术支持人员的学习曲线是最小的。可追溯性（Traceability）、审计和日志记录至关重要。当系统管理员很开心时，大家都会开心（尤其是老板）。

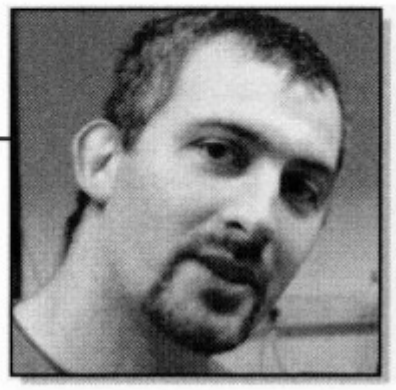
作者简介：

门西蒂西·卡斯珀（Mncedisi Kasper）是 Open Xcellence ICT Solutions 公司的技术和战略总监，该公司位于南非，专长是企业应用集成和 SAP（ABAP/XI）咨询。

有舍才有得

Prepare to Pick Two

比尔·德·霍拉（Bill de hÓra）



有时，接受某种约束或放弃某个特性，可带来更好的架构，这种架构在构建（build）和运维（run）上都会更加简单，而且成本更低。假设期望的理想特性有 3 种，试图定义和构建支持所有这 3 种特性，则可能一无长处。

一个著名的例子是布鲁尔猜想（Brewer's conjecture），也被称为一致性（Consistency）、可用性（Availability）和分区（Partitioning），即 CAP 定理。该定理指出，在分布式系统中通常期望的 3 个特性，即一致性、可用性和分区容错性（partition tolerance）是无法同时获得的。试图同时拥有三者，将大幅增加工程费用，显著提高复杂度，也无法真正获得预期效果和实现业务目标。如果数据必须是可用和分散的，强求一致性会不断增加成本，最终将无法达成目标。同样，如果系统必须是分布和一致的，则确保一致性首先将会导致延迟和性能问题，因为系统在各部分未同步之前无法提供服务，它最终是无法使用的。

人们常常认为有某个或多个特性是不容侵犯的（inviolate）：数据是不能重复的，所有的写入都必须是事务性的，系统必须 100%可用，调用必须是异步的，必

须没有单点故障，所有一切必须都是可扩展的，等等。除天真幼稚之外，如果将这些特性奉为不容侵犯的教条，你将无法对手头的问题开展真正的思考。我们要谈论的是架构设计上的偏移（architectural deviation），而不是原则层面上的设计，我们惑于教条主义的统治束缚久矣。相反，我们应该问，为什么必须要持有这些特性？这样做可以获得什么好处？何时才期望拥有这些特性？如何才能打破系统成规以达成更佳的效果？永远不要放弃质疑，因为架构设计的教条往往从根基上削弱了交付能力。这种权衡不可避免，它是软件开发中最难以应对的事情之一，这不只对架构师而言，即使对开发人员和利益相关者也是如此。但是，我们应当珍惜这种权衡，它们远胜毫无限制的可选项，并且，接受一些权衡，往往能产生富有创造性和创新性的结果。

作者简介：

比尔·德·霍拉（Bill de hÓra）是 NewBay Software 的首席架构师，专注于大规模 web 和移动系统方面的工作。他是 Atom 发布协议的共同编辑者，此前他服务于 W3C 的 RDF 工作组。在 REST 风格和消息传递（message-passing）架构与协议的设计上，他是公认的专家。

先考虑原则、公理和类比，再考虑个人意见和口味

Prefer Principles, Axioms, and Analogies to Opinion and Taste

迈克尔·哈默（Michael Harmer）



个人意见和口味上的分歧，常常会演变为政治性的争论，其中就会出现动用权威来赢取争论的可能。然而，只要底层原则十分清楚，分歧便可为深入的理性探讨开辟道路，同时又规避了与个人有关的问题。可能根本无需牵扯到架构，便可解决分歧。

在架构实现及整个过程中，原则和公理也确保了架构上的一致性。一致性通常会成为问题，尤其在跨越多种技术并将长期存在的大型系统中。清楚的架构原则，能够使那些不熟悉某项特别技术或组件的人，明白其中的缘由，更透彻地理解他们本不熟悉的技术。

作者简介：

迈克尔·哈默（Michael Harmer）在软件界已经工作了 16 年，担任过软件开发者、团队带头人、架构师，首席工程师和实践经理（practice manager）等多种角色。

从“可行走骨架”开始开发应用

Start with a Walking Skeleton

克林特·尚克（Clint Shank）



为了实现、验证和不断发展应用架构，一个非常有用的策略，便是从阿利斯泰尔·考克伯恩（Alistair Cockburn）所谓的“可行走骨架”开始。“可行走骨架”是对系统的最简单实现（a minimal implementation），它贯串头尾（end-to-end），将所有主要的构架组件连接起来。从可工作的最小系统开始来训练全部的通信路径（communication path），可以带来“正朝着正确方向前进”的信心。

骨架一旦就绪，就该进入“健身”环节了。通过“全身锻炼”使系统不断成长起来，即增量实现，逐步增加贯串一体的（end-to-end）功能。目标是要在培育骨架成长的过程中，保持系统一直运行可用（keep running）。

架构的调整是颇为困难和昂贵的，而且历时越久，系统架构也会变得越为庞大。能够越早发现错误越好。“从可行走骨架开始”这一方法，能够创建很短的反馈回路，可以更快速地对系统进行调整，以迭代的方式按优先级列表上的次序满足业务需求，而且这种优先级列表上的质量属性在运行时是可量化衡量（runtime-discernable）的。对架构所做的假设也可以较早地得到验证。由于在早期阶段就发现了问题，这时在实现上还投入不多，架构的演化发展会更为容易。

系统越庞大，使用这一策略就越显重要。对于小型应用程序，一名开发人员就可以从头到尾相对迅速地实现整个功能，但在较大的系统中，这样做就变得不切实际了。大型系统，通常都会由多名开发人员构成一个开发团队——甚至在整个实施过程中，会有多个可能同时分布在多处的团队——来共同完成。因此，对大型系统开发而言，更多的协调工作是必不可少的。当然，开发人员的效率肯定会有差异。有些开发人员能够在很短的时间内做完许多工作，而其他人可能花了很多时间却成果寥寥。在项目中越是困难大、耗时多的任务，越早完成越好。

从“可行走骨架”开始，保持系统一直运行可用，增量式地进行培育，使其逐步成长。

作者简介：

克林特·尚克（Clint Shank）是 Sphere of Influence 公司的开发人员、顾问和导师，该公司为商务合作客户提供软件设计和工程服务。

数据是核心

It Is All About The Data

保罗·W·霍默（Paul W. Homer）



软件开发人员最初一般将软件理解为是由命令、函数和算法构成的系统。在学习构建软件的方法时，从面向指令的视角（instruction-oriented view）来认识软件确有帮助，但也正是这一视角，在开发人员尝试创建较大规模的系统时，开始造成阻碍。

如果稍稍后退站远一点看，计算机只不过是能访问与操作一堆数据的时髦工具而已。对于了解管理庞大系统的复杂性，数据的结构处于核心地位。数以百万计的指令有其固有的复杂性，但是，我们可以把注意力集中于底层那些小得多的基本数据结构集上。

举例而言，如果想了解 Unix 操作系统，通过源代码逐行挖掘是不大可能奏效的。但是，如果你读过一本讲解 Unix 内部数据结构的书，由于数据结构与进程处理、文件系统等关系密切，因此，如果你读过了一本概括 Unix 内部数据结构的书，便可更好地了解 UNIX 在底层是如何运行的。从概念上来看，数据要比代码更加精炼，也更好理解。

代码在计算机中运行时，底层数据的状态不断发生变化。在某种抽象意义上，可以认为任何算法都只是数据从一个版本到另一个版本的迁移（transformation）而已。我们可以把功能视为是“更多定义良好的迁移所构成的集合”（a larger set of well- defined transformations），在多个版本间连续推动数据的流动。

即使对于最复杂的系统，通过这种面向数据的视角（data-oriented perspective），即通过底层信息的结构整体来看待系统，也可以将之缩减为细节的有形集合（a tangible collection of details）。为了了解复杂系统是如何构建和运行的，必须降低其复杂性。

数据在大多数问题中处于核心地位，业务领域问题经由数据蔓延到代码中。举例而言，大多数关键算法往往易于理解，频繁变化的反而是结构及数据之间的关系。像升级这类运维问题，如果影响到数据，解决起来也会相当困难。要改变代码和行为不是大问题，将之发布即可，但是要将数据结构从老版本迁移到一个新版本，可能需要付出巨大努力。

诚然，软件架构中的许多基础问题确实和数据相关。系统是否在正确的时间收集了正确的数据？谁能够看到数据或修改数据？如果数据已经存在，其质量如何？增长速度如何？如果数据以前不存在，该如何设计数据的结构?何处是数据的可靠来源？从这种视角看，一旦数据已进入系统，剩下的唯一问题便是：是否已经存在查看和编辑特定数据的方法？还是有待增加这些方法？

从设计角度来看，大多数系统的关键问题，就是要在正确的时间从系统中获得正确的数据。从这种角度出发，对数据执行迁移（transformation）操作，便是获取数据、运行功能，然后保存运行结果。为了提供功能，大多数系统其实并不需要特别复杂的底层，它们只是需要积累起越来越多的数据。最先可以看到的确实是系统的功能，但是，只有数据真正构成了每个系统的核心。

作者简介见第 109 页。

确保简单问题有简单的解

Make Sure the Simple Stuff Is Simple

查德·拉·瓦因（Chad La Vigne）



软件架构师解决了很多非常困难的问题，但也会去解决一些相对容易的问题。对于简单的问题，不要使用复杂的解决方案。这个建议听上去显而易见，但要遵循却并不容易。软件设计者都是聪明人，真的很聪明，但是出于炫技心理，很容易陷入“杀鸡用牛刀”的误区（simple problem-complex solution trap）。如果发现自己正在设计一个非常聪明的解决方案，也许应该对此保持警醒自觉（self-aware），停下来想想：解决方案是否对症？如果答案是否定的，则需要重新考虑设计中的各个选项。对简单问题要保持简单的解（Keep the simple stuff simple）。架构师展示才华的机会多的是，只要有真正的难题出现，总有这样的机会。

但这并不是说不要追求漂亮的解决方案，而是指，如果我们的任务是设计这样的系统：它只需要支持销售单一种类的基于 SKU（Stock Keeping Unit，最小存货单位）的商品，那么，设计成支持可动态配置产品层次结构的系统可能就是糟糕的主意。

为复杂解决方案付出的直接成本可能看上去很小，但是其潜在成本要大得多。和开发层面的过度工程一样，架构上的过度工程（over-engineering），会导致许多问题，但其带来的负面影响则往往会成倍增加。错误的架构设计决策会使系统实现困难且不易维护，最糟糕的是，实现是无法逆转（reverse）的。在往前做出超越系统实际需求的架构决策时，不妨自问：照此实现之后，如果要再退回去会多么困难？

为有问题的解决方案付出的代价，不仅仅只是在实现和维护上。在简单问题上耗费了过多时间，则留给解决复杂问题的时间相对就少了。不当的架构决策陡然间导致了范围的蔓延，往项目中增加了不必要的风险。如果你能保证其他人也不会这么做，你的时间利用率就会大大提升。

架构师会从主观的判断或潜在不确定需求出发，产生调整解决方案的强烈冲动。请记住一点：试图猜测未来的需求时，错的概率是 50%，错得很离谱的概率是 49%。今天只需要解决今天的问题就好。把应用发布出去，从反馈中生成真实的需求。由于已经创建了简单的设计，当真实需求到来时，要将之整合进来就会更加容易。如果刚好运气不错，先前猜测的需求在下一个发布版本中变成了真实的需求，那现在更是成竹在胸了。与之前不同的是，现在可以估算出更合适的时间分配了，因为这次它已经变成真实的需求。不知不觉间，团队就会获得“能做出精准的预估，按时完工”的好名声了。

作者简介见第 111 页。

架构师首先是开发人员

Before Anything, an Architect Is a Developer

迈克·布朗（Mike Brown）



你听说过谁是法官但没当过律师，或者谁是外科主任但没当过外科医生的事情吗?人们认为法官和外科主任是律师和外科医生职业生涯的顶峰。但即使在已经达到职业顶峰之后，这些已身居要位者仍然需要继续跟进各自领域内的最新发展。对软件架构师，应该也秉持同样的标准。

不管解决方案设计得如何优秀，决定实现能否成功的最重要因素之一是让开发人员愿意接下任务。让开发人员肯接活最快捷的办法，是获得他们的尊重和信任。我们都知道获得开发人员信任的最快捷方式：你的代码就是你的资本。如果能向开发人员表明，你不是那种只会吹得天花乱坠但不会编程的空想家，你就会少听到许多报怨。假若要把数据显示在页面上，你命令“把数据集绑定在数据格子上，这样速度快些”，他们就不会说你在强迫他们走冤枉路。

虽然不是工作所要求的一部分，我仍然会经常去处理一些比较复杂的任务。这样做有两个目的：第一，这是一种乐趣，而且还能帮我做到宝刀不老；第二，这有助于向开发人员表明，我不是碰到麻烦时只会干吹烟圈却束手无策的那类架构师。

作为架构师，主要目标应该是创建可行、可维护的解决方案，当然，也一定要能够解决当前的问题。其中，要知道解决方案中什么是可行的，就要求架构师拥有相关知识，能够实际参与到解决方案的开发活动中去。因此我建议，如果是你做的系统设计，你也应该能够编程实现自己给出的设计。

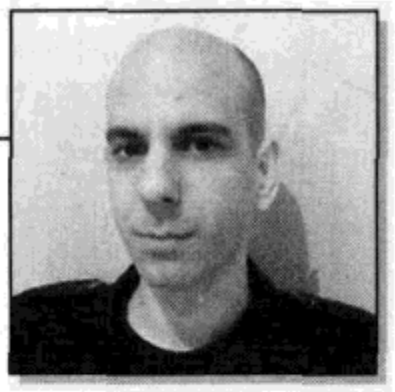
作者简介：

迈克·布朗（Mike Brown）是 Software Engineering Professionals 公司（*http://www.sep.com*）的杰出软件工程师。他在 IT 界拥有 13 年的从业经验，其中包括 8 年的企业解决方案开发经验，覆盖了广泛的垂直市场。他是印第安纳波利斯 Alt.NET 用户组的创立者、WPF Disciples 社团的创始成员，以及即将启动的 Indy Arc（印第安纳波利斯式赛车运动）专业用户组的组织者。

根据投资回报率（ROI）进行决策

The ROI Variable

乔治·马拉米迪斯（George Malamidis）



我们对项目所做的每一个决策——无论是与技术、过程，还是与人相关——都可以看作一种投资形式。投资是和成本联系在一起的，成本并非单纯只有货币一种形式。之所以进行投资，是相信它们最终能带来回报。老板发员工薪水，是期望此项投资将会对他们的事业产生积极的影响。开发团队决定遵循某种专门的开发方法学，是期望它能够给团队带来更高的生产力。选择投入一个月的时间重新设计应用程序的物理架构，是相信这将有利于长期运维。

回报率（rate of return），也称为投资回报率（Return On Investment, ROI），是衡量投资是否成功的指标之一。举例来说，“我们预计，投入更多时间用于编写测试，则下一个产品发布版本中的缺陷将会少很多”。在这个例子中，投资成本是用于编写测试的时间。所获得的，是将来在修复缺陷上可以节省下来的时间，以及提高软件体验后的客户满意度。假设目前一周中的 40 个工作小时，有 10 个小时被用于修复缺陷。如果每周投入 4 个小时用于测试，预计可以把用于修复缺陷的时间减少到每周 2 小时，这样将可以有效节省 8 小时投入到其他地方。从修复缺陷中省下的 8 小时，除于用于测试的 4 小时，预期的投资回报率为 200%。

虽然不必用经济收益来衡量一切事物，但投资总应该产生增值。在当前项目中，如果上市时间对投资方是至关重要的，那么，由于需要经历漫长的前期设计（upfront design）阶段，获取“无懈可击的（bulletproof）架构”也许就不如生产“可迅速发布的 alpha 版本”，后者能够提供更有吸引力的投资回报率。只要迅速发布可用版本，我们就能根据用户反应进行适当调整，掌握对未来方向发展和项目成功的关键决定因素。但另一方面，由于没有进行整体的规划，在需求上升时，可能无法很容易地对应用进行伸缩扩展，这样又可能要为之付出相应代价。通过对比成本和预期利润，可以算出每个决策选项的投资回报率，将 ROI 作为多个决策选项之间的评选基准。

将架构决策视为投资，并将相关的回报率也一并考虑在内。在判断每个决策选项是否务实（pragmatic）或恰当时，这种方法很有用。

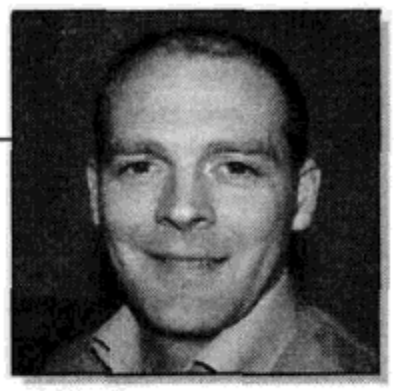
作者简介：

乔治·马拉米迪斯（George Malamidis）是位软件工程师，为位于伦敦的 TrafficBroker 公司工作。此前，他是 ThoughtWorks 公司的首席顾问和技术主管。他曾帮助在多个领域中交付关键性应用，范围覆盖网络系统到银行业务到 Web2.0。

一切软件系统都是遗留系统

Your System Is Legacy; Design for It

戴夫·安德森（Dave Anderson）



即使系统十分前沿，采用了最新的技术开发而成，但对接手它的下一个人而言，它也会是遗留系统。必须应对这种情况！在今天，软件很快便会过时，这已经成为软件的天然属性。如果系统能够作为产品存活下来，哪怕只是数月时间，都必须承认一点：负责维护工作的开发人员肯定要对软件进行缺陷修复，这是不可避免的。这引出如下几个问题。

- 清晰性（Clarity）：各个组件和类的角色一定要十分清楚。
- 可测性（Testability）：系统易于验证吗？
- 正确性（Correctness）：结果和设计或期望的一致吗？要避免那种虽然迅速但手段低劣（quick and nasty）的修复方式。
- 可跟踪（Traceability）：假设恩尼（Ernie）负责修复紧急缺陷，但此前他还从未看过代码，他能马上跳入产品中诊断出故障所在并立马解决吗？还是需要 8 周的交接时间？

假设有另外不同的团队打开了代码库，他们很容易便可了解到当前在做什么，这是优秀架构的基础。无需对架构进行过度的简化或为之准备面面俱到的记录文档；好的设计会以多种方式说明自身。在生产环境中的系统行为也会说明其设计，举例而言，依赖关系十分丑陋的架构，其行为往往看起来就像是笼中的困兽，到处受限。替那些要进行缺陷调试的（通常都是较为初级的）开发人员着想一下吧。

在软件界，“遗留系统（legacy）”不是好词，但实际上，所有的软件系统都不应该排斥这个标签。它并不是一件坏事，因为这或许也表明了你的系统久经考验，符合预期，具有商业价值。任何软件系统，如果从未曾被称为“遗留系统”，那也许它早在发布前就已遭到被抛弃的厄运了，这可不是表明软件架构取得成功的迹象。

作者简介：

戴夫·安德森（Dave Anderson）是位于贝尔法斯特的 Liberty IT 软件公司的首席软件工程师，Liberty IT 为财富 100 强公司——利宝（Liberty Mutual）提供 IT 解决方案。戴夫曾在横跨多个不同行业和国家的前沿 IT 公司呆过，拥有超过 10 年的软件从业经验。

起码要有两个可选的解决方案

If There Is Only One Solution, Get a Second Opinion

蒂莫西·海伊（Timothy High）



你可能听过这条规矩。如果你是一名经验丰富的架构师，你肯定知道这点：对于某个问题，如果只考虑了一个解决方案，那你就有麻烦了。

软件架构是要在所有给定的约束条件下，寻找到解决问题的最佳方案。期望第一个解决方案即满足全部的需求和约束，几乎是不可能的。一般来说，必须根据优先级次序进行权衡，选择最符合需求的解决方案。

如果对手头的问题只有一个解决方案，这意味着将没有进行折衷权衡的余地。这个唯一的解决方案很可能无法令系统投资方满意。这也意味着，如果由于商业环境瞬息万变而导致优先级次序发生转变，系统很可能没有空间来适应新的需求。

这种情况即使有——也绝少是真地由于缺乏可选方案而造成的，它更可能是架构师缺乏特定问题域的经验所致。如果事实确是如此，别费力气，赶紧让更有经验的架构师助你一臂之力。

如果依照习惯设计架构，这个问题将更不易为人所察觉。架构师也许已熟稔于某种单一风格的架构（例如，三层或多层的客户机-服务器系统），却没有充分认识它不适用的情况。如果发现自己在还没有对比其他方法之前，就已经想当然地给出了解决方案，那么，停下来，向后退一步，问问自己，是否能够想出另一种方法。如果不能，你可能需要帮助了。

我的一个朋友，曾是一家成长中的小型互联网创业公司的技术负责人。随着用户群的不断扩大，他们系统的负载与日俱增，性能也每况愈下，公司开始渐渐失去来之不易的用户。

于是，老板问他，“我们该如何提高系统性能？”

我的朋友回答：“买一台更大的机器！”

“还有什么其他可做的？”

“嗯……据我所知，只有这样了。”

我的朋友被当场解雇了。当然，这个老板是正确的。

作者简介见第 103 页。

理解变化的影响

Understand the Impact of Change

道格·克劳福德（Doug Crawford）



好的架构师能够将复杂性降到最低限度，他在解决方案中给出的抽象，应该能够为更高的层次提供坚实基础，同时，还应该能足够务实地应付未来的变化。

优秀的架构师能够深刻理解变化带来的影响，这种影响不仅限于彼此隔离的软件模块之间，而且包括人与人之间，以及系统与系统之间。

变化有多种不同的表现形式：

- 功能需求的变化。
- 可扩展性需求的演进。
- 系统的接口被修改。
- 团队人员流动。
- ……

软件项目中变化的广度和复杂性，是无法提前看得清楚，企图事前消解所有潜在冲突（bump）是徒劳无功的。但是，在确定沿途遭遇的冲突和项目成败之间的关系中，架构师扮演着关键角色。

管理变化并非架构师的职责，但架构师要确保变化是可控的。

举例而言，高度分布式的解决方案，横跨了许多应用程序，依赖多种中间件（middleware）将各个部分胶合在一起。如果没有对依赖集（set of dependencies）进行准确跟踪或以某种可视化模型表现出来，业务流程的变化可能会造成重大危害。如果变化会影响数据模型或破坏现有接口，其对下游的影响会尤为巨大。另外，现存的长时间运行中的有状态事务（long-running stateful transaction），在老版本流程下也必须要能成功完成。

这个例子可能有点极端，但高度集成的解决方案现在正是主流。架构师需要在可用的集成标准、框架和模式中做出选择，这个例子无疑可以说明这点。为了能给客户提供一贯的支持，理解这些外围系统的变化将带来的潜在影响，显得至关重要。

幸运的是，有许多工具和技术可以用以减轻变化的影响：

- 进行小规模的增量渐变。
- 构建可重复运行的测试用例，并经常运行。
- 让测试用例更易编写。
- 跟踪好依赖关系。
- 系统性（systematically）的行动，根据反馈信息作出进一步反应。
- 自动化重复的任务。

架构师必须评估变化对项目范围、时间和预算各个方面产生的影响，并准备好花较多时间在那些受影响最大的区域，即“沿途的冲突（a bump in the road）”上。通过衡量风险（measuring risk），可以知道该把宝贵时间花在哪里。

降低复杂性很重要，但降低复杂性并不等于简化处理。务必清楚认识解决方案中变化的类型和将带来的影响，对中长期而言，这样做带来的回报是无法估量的。

作者简介：

道格·克劳福德（Doug Crawford）为南非一家电信公司管理中间件开发团队。近 10 年时间中他解决了许多疑难杂症，并一直密切参与应用集成项目，行业范围涉及广告、企业贷款、保险和教育。

你不能不了解硬件

You Have to Understand Hardware, Too

卡迈尔·威克拉玛纳亚克（Kamal Wickramanayake）



对于许多软件架构师，硬件容量规划问题是一个超出其舒适区的主题，但它的确是架构师工作的重要组成部分。软件架构师常常无法正确考虑硬件因素，有多种原因，但大多和缺乏对硬件的了解及需求不清晰脱不了干系。

之所以忽视对硬件的考虑，其首要原因是，架构师把全部精力都花在软件上，所以往往就忽略了硬件上的要求了。除此以外，由于使用高级语言和软件框架，软件架构师和硬件就离得更远了。

需求不清晰也是一个因素，因为需求可能会发生变化，也可能没有被充分透彻地理解。随着架构的演进，对硬件的考虑因素也将随之改变。此外，客户可能也不了解或无法预测用户群规模和系统的使用动态。而且，硬件本身也在不断发展，以前掌握的硬件知识并不适用于今天。

如果没有硬件方面的专业知识，预估待开发系统的硬件配置是非常容易出错的。为了弥补这方面的不足，有些软件架构师会使用较大的安全系数。这种安全系数一般不是基于客观评估的，也无法从什么方法学中找到理论依据。在大多数情况下，这将造成基础设施的容量规划过度，即使在请求高峰期都无法充分利用基础设施的能力。最终，客户的钞票被浪费在超出系统实际所需的硬件上了。

对缺乏硬件规划能力最好的防御措施，是和基础设施架构师（infrastructure architect）紧密合作。不同于软件架构师，基础设施架构师是硬件容量规划方面

的专家，而且他们也应该是团队的一部分。然而，并不是每一个软件架构师都可以奢侈地配备一名基础设施架构师和他一起工作。这种情况下要进行硬件规划时，软件架构师可以采取一些措施来减少错误的发生。

可以借助自己过去的经验。你已经实现过一些系统，所以肯定已经拥有一些硬件容量规划的知识，哪怕当时是事后才考虑的也没关系。还可以和客户就这个主题进行讨论，说服他们为硬件容量规划预留资金。比起采购超出实际所需的硬件，为容量规划留出预算是更为经济的。此种情况下，水平伸缩能力（horizontal scalability）是关键，你可以在需要时才添加硬件，而无需一开始就过量采购。为使水平伸缩策略起效，软件架构师需要不断评测系统容量，隔离软件组件，使它们在性能可预知的（performance-predictable）环境下运行。

硬件容量规划是和软件架构同等重要的事情，不管身边是否有基础设施架构师，都要将之安排在第一优先级。架构师既要负责连接业务需求和软件解决方案，也要负责连接硬件和软件。

作者简介：

卡迈尔·威克拉玛纳亚克（Kamal Wickramanayake）是位 IT 软件架构师，住在斯里兰卡。他通过了 The Open Group 的 TOGAF（开放组组架构框架，The Open Group Architecture Framework）认证。

现在走捷径，将来付利息

Shortcuts Now Are Paid Back with Interest Later

斯科特·麦克菲（Scot Mcphee）



长远看来，系统维护将比项目初期的开发消耗更多的资源。进行系统架构设计时，牢记这点非常重要。在项目开发初期走捷径，可能会以日后付出高昂的维护费用为代价。

例如，你可能觉得单元测试并不直接产生价值，于是就让开发人员跳过这些严格的测试工作。这将导致所交付的系统在未来更难修改，而且在修改时信心不足。即使只做了一点修改，也需要对系统做大量的手动测试，这将导致系统极为脆弱，维护成本不断攀升，其设计也将远逊于经过全面测试的系统，更别说和使用了测试先行（test-first）的设计相比了 。

如果以为“使用既有系统多少可以节省成本”，于是试图生搬硬套某个既有系统，则是犯了严重的架构错误。例如，你可能会将 BPEL 组件和数据库触发器（trigger）结合起来开发异步消息系统。这样做或者坚持要这样做也许是出于便捷，也可能是因为自己或客户对这样的架构更熟悉。但是，在需求明确后，最先确定选用、不可或缺的应该是消息架构（messaging architecture）。项目之初的糟糕决策，会让重新设计系统架构以满足新需求的费用变得更为高昂。

除了避免在开发初期走捷径，发现有不当的设计决策时就要尽快修正，这点也很重要。设计不当的特征可能会成为后续特征的基础，将来需要花很高的成本来更正。

例如，假使发现为实现某些潜在功能而选用了不适合的类库，那就赶紧把它们替换掉。否则，为使它们适用于新的需求，将不得不设计更多的抽象层次（layers of abstractions），以弥补前一层次的不匹配性。这样，每增加一层，都无异于将自己置身于一团乱麻之中，剪不断理还乱，系统终将陷入无法维护的境地。

碰到架构问题或设计缺陷，作为架构师，一定要坚持成本还很低廉时就动手。搁置越久，为之付出的利息也将越高。

作者简介：

斯科特·麦克菲（Scot Mcphee），来自澳大利亚的软件开发者和架构师，拥有超过 15 年的程序设计和开发经验。在过去 8 年的工作中，他主要使用 J2EE 系列技术。

不要追求“完美”，“足够好”就行

"Perfect" Is the Enemy of "Good Enough"

格雷格·纽伯格（Greg Nyberg）



软件设计师，尤其是架构师，在评估针对某个问题的解决方案时，会倾向于考虑它是否优雅完美。我们在查看设计或实现时，仿佛自己是选美比赛的评委，会立即找出其中的微小瑕疵，而这些瑕疵只须经由一些调整或迭代重构便可消除。对领域模型中是否存在可被移入基类之中的共同属性或函数过于关注；当在多个实现中发现存在重复的服务时，就大声疾呼一定要把它变为 web 服务；对“从缓存中得到的” 查询和非唯一性索引，则指摘不止。

我的建议是：不要屈服于企图使设计或实现达到完美的诱惑！把目的设定在“足够好”就行，当已经达成目标时，就停下来。

你可能会问，究竟什么是“足够好”？“足够好”指的是，剩余的不完美之处，对系统的功能、可维护性或性能不会产生任何有深远意义的影响。架构和设计协调一致；系统的实现正确可用，并符合性能需求；代码整洁简明，文档化良好。还可以做得更好吗？当然可以，但这样已经足够好，所以就到此为止了吧。可以宣布设计胜利完成，然后转入下一个任务了。

在我看来，在设计和实现上追求完美，会导致过度设计（overdesigned）和模糊混乱的解决方案，最终使系统难以维护。

本书中已有不少主题都在提醒设计师，要注意避免不必要的抽象或复杂性。为什么保持简单会这么困难呢？因为我们在寻求完美的解决方案！架构师无法忍受设计中可感觉到的不完美，定要除之而后快，但正是这种冲动，向已可工作的简洁的解决方案中引入了额外的复杂性！

请记住，应用程序开发不是选美大赛，因此，停止吹毛求疵的做法，不要再浪费时间追求尽善尽美。

作者简介：

格雷格·纽伯格（Greg Nyberg）现在是名独立的 J2EE 顾问，在设计、构建、测试和部署大型高容量交易处理型应用程序，如预订系统、呼叫中心和消费者网站方面，拥有 18 年的经验。他是《WebLogic companion workbook for Enterprise JavaBeans，第三版》（O'Reilly）的作者，以及《Mastering WebLogic Server》（Wiley）的主要作者。

小心“好主意”

Avoid "Good Ideas"

格雷格·纽伯格（Greg Nyberg）



“好主意”会杀死项目。有时候杀伤力很快见效，但更常见的症状是：因屡屡错过里程碑和不断攀升的缺陷数量，项目苟延残喘，最终不治而亡。

你知道我指的是什么类型的“好”主意：那种诱人的、不用想都知道的（no-brainer）、外表无辜、以为不可能会产生伤害（couldn't-possibly-hurt-to-try）的那种“好”主意。通常在项目进展到一半而似乎一切看起来都挺好——形势和进度都在循序渐进，初步测试进展顺利，落地（rollout）日期看起来可靠无误——的时候，项目团队中有人会冒出这些想法。生活很美好。

有人冒出一个“好”主意，你默许了。于是，为了利用 Hibernate 的新特性，项目中需要改用新的版本；由于开发人员向用户展示了很酷的 AJAX 效果，就被要求在一些页面中增加 AJAX 的实现；甚至为了利用 RDBMS 的 XML 功能，还要对数据库进行重新设计。你最初告诉项目经理说需要几个星期时间来实现这个“好”主意，但最终，受影响的代码要比原先预期的多得多，而开发进度已开始滞后。加之，最初允许把“好”主意放进来时，就已允许如谚语中所说的“骆驼鼻子”（译注 1）伸到帐篷里来了，很快，“好”点子就忽然间如雨后春笋般涌现，让你欲拒不能（骆驼很快就会睡到你的床上了）。

“好”主意真正隐蔽的邪恶之处（insidious thing）是，它们是“好”的。糟糕的坏主意，每个人都看得透，都会拒绝。“好”主意是漏网之鱼，它将导致范围膨胀，复杂度上升，竭力把和业务需求无关的东西塞入应用中，这纯粹是浪费精力。

译注 1：有一句古老的阿拉伯谚语：骆驼一旦把鼻子伸进帐篷，马上它的身子也会跟着进来。(If the camel once gets his nose in the tent, his body will soon follow.)“骆驼的鼻子”，是一个隐喻，指一旦允许一些不期望但很小的情况发生，后面就会招致巨大而无法避免的更糟情况。

如果出现下面这些关键词，要小心了：

- “如果……，会更酷。”实际上，任何语句如果带有“酷”字，都是危险信号。
- “嘿，他们刚刚发布了 YYY 框架的 XXX 版本。我们应该马上升级！”
- “由于我们在使用 ZZZ，你知道，我们确实应该重构 XXX……”
- “XXX 技术真的很强大！也许我们可以把它用于……”
- “嘿，<某某>，我一直在思考这个设计，我有一个想法！”

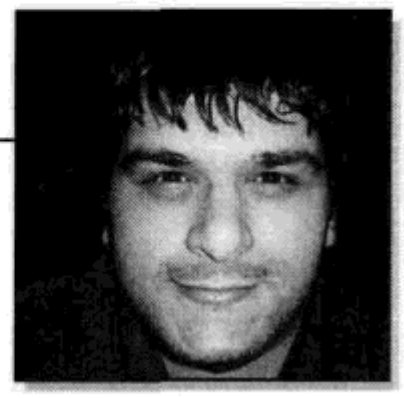
好吧，好吧，也许我对最后那条有点过于吹毛求疵了。但是，务必小心那些“好主意”，它可能会杀死你的项目。

作者简介见第 141 页。

内容为王

Great Content Creates Great Systems

朱宾·沃迪亚（Zubin Wadia）



我见过无数这样的设计，它们大都永无止境地强调需求、设计、开发、安全及可维护性，但从未关注系统的真正要点——数据。对于基于内容的系统（content-based system），数据尤其重要，对它们而言，所谓“数据”，即是以非结构化或半结构化内容的形式交付的信息。所谓优秀的内容，指的是其内容之间互相关联，而不是空洞割裂。

内容为王。内容即网络，即界面。在联系日益紧密的今日世界，内容质量（content quality）很快就成为了成败的关键。FaceBook 对 Orkut，Google 对 Cuil，NetFlix 对 BlockbusterOnline……列表还可以更长 。这一连串的较量，都说明内容是决胜的关键。或许有人会争辩说，内容相关方面不是软件架构师应该操心的问题，但我认为在未来 10 年，这样的观点一定会被驳倒。

在新系统的设计过程中，必须留出一部分精力专心考评内容库（content inventory）。仅仅设计出有效的领域模型、对象模型或数据模型是不够的。

可以根据以下标准，分析所有可用的内容，评估它们的价值：

- 是否可以获得足够多的内容？如果不是，我们如何去获取关键的内容？
- 内容够新鲜（fresh）吗？如果不是，该如何提高交付率（delivery rate）以保证内容的新鲜度？

- 是否已利用了所有可能的内容渠道？RSS 订阅、电子邮件和纸质表单，这些都是可利用的渠道。
- 是否建立了有效的输入流，可将内容源源不断地引入系统中？一方面要能识别出有价值的内容，另一方面要定期获取和更新内容，两者结合才能赢得胜利。

没错，系统的成功取决于其内容。在设计过程中，要对内容价值的评估给予足够重视。如果评估结果不如人意，要向利益相关者反馈那些不足之处并给出建议。我见过许多系统虽然完成了所有的合同义务，也满足了各项需求，但仍以失败告终，其原因就是一个相当明显的因素被忽视了：优秀内容成就优秀系统，内容为王。

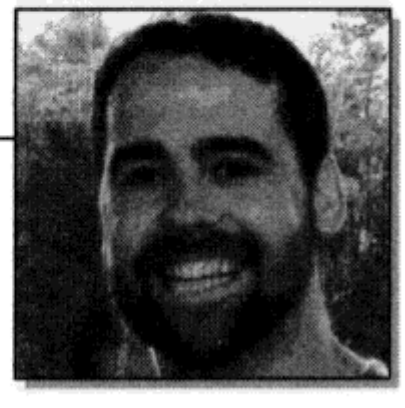
作者简介：

朱宾·沃迪亚（Zubin Wadia）同时兼任 RedRock IT Solutions 公司的 CEO 和 ImageWork Technologies 公司的 CTO。他具有丰富的软件开发背景，使用过多种不同开发语言，包括 Basic、C、C++、Perl、Java、JSP、JSF、JavaScript、Erlang、Scala 和 Ruby 语言。他目前主要关注设计业务流程自动化（business process-automation）解决方案，以帮助全球财富 500 强公司和美国政府机构提高工作效率。

对商业方，架构师要避免愤世嫉俗

The Business Versus the Angry Architect

查德·拉·瓦因（Chad La Vigne）



在架构师的职业生涯中，我们发现碰到的问题许多都是反复出现的。尽管项目和行业可能会有所变化，但许多问题都是相似的。这时，我们可以凭借经验，迅速提供出许多解决方案，从而可以留出更多时间用以解决具有挑战性的问题，享受其中的乐趣。我们对自己的解决方案充满信心，而且最终也总能做到如期交付。我们感觉自己已经达到了某种内在平衡（homeostasis）的完美境界。但恰恰这时，也是最容易犯下巨大错误的时候——自我感觉过于良好，渐渐开始忘记了倾听。自认为自己无论是对技术还是其他方面，都已是了如指掌，而那些远不如你的人居然敢提出反驳意见，于是乎，你便以冷嘲热讽、极不耐烦甚至无名之火相对，这时通常便会做出糟糕的决策。

过度自信，会让你在业务领域头破血流。黑犀牛（Black Rhino）是架构师这一职业的最好写照。业务是架构师职业存在的原因。这么说可能会令人不快，但我们绝不能无视这一事实。为业务服务是我们的生存之本，而不是相反地让业务为我们服务。雇主期望我们能够解决问题，倾听和了解雇主的业务，是我们必须掌握的最为关键的技能。是否发现自己已经没有耐心听业务分析师说完后再表达自己的观点？如果确是这样，你很可能根本就不明白业务分析师要说的是什么。想要业务领域专家尊重你，首先要同样地尊重他们。如果他们都认为你无法接近，那么就没人愿意和你沟通了。架构师本该作为沟通的催化剂，如果他们开始避开你，那表明催化剂已经失效，你自己的项目也因而受到影响。记住，自说自话的时候，是无法听进任何新东西的。千万不要在一开始就以为自己是最聪明的人，而其他人的话毫无价值不值一听。

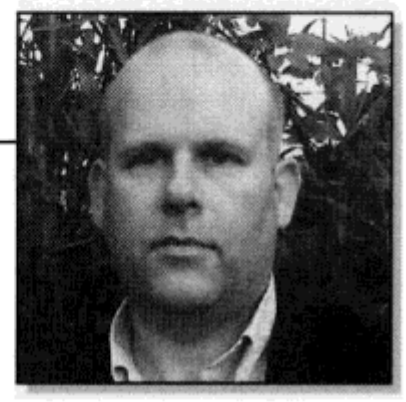
在沟通时，我们经常会对听到的业务运营方式提出异议。这很好。我们可以提出改进建议，的确应该这样做。但是，如果一天沟通下来，你只是在不断表达你是如何不赞同当前的业务运营方式，没有提出任何建设性的改进建议，那就太糟糕了。不要让自己成为愤世嫉俗的“天才”，总是以一副自作聪明、居高临下的语调，力求证明公司当前的运营是多么糟糕不堪，以期触动业务方。业务方不会受到一丁点触动的。他们以前早就碰到过这样的家伙，事实上，他们并不喜欢这样的人。成为优秀架构师的秘诀之中有一个关键要素，那便是：要对工作满怀激情（passion），但不要是那种带着愤怒和火气的“激情”。要学会能够带着不同意见继续前进。如果发现分歧过大而自己总是与业务方争吵不休，那么，换一家对你而言更易合作的公司来为他们工作。总之，要找到能和业务方建立良好关系的方法，不要让自我（ego）破坏这种关系。这会使你成为更快乐、更高效的架构师。

作者简介见第 111 页。

拉伸关键维度，发现设计中的不足

Stretch Key Dimensions to See What Breaks

斯蒂芬·琼斯（Stephen Jones）



应用程序的设计轮廓，最初是基于特定的业务需求、所选择的技术或现有的技术、性能要求、预期的数据量，以及构建、部署和运营上可用的财务资源。无论采用什么样的解决方案，都要求能满足或超越当前环境下的要求，成功运行起来，否则这就不成其为解决方案了。

现在来拉伸解决方案中的关键维度，看看哪些方面会遭到破坏。

使用这种检查方法，可以发现系统设计的限制。这种限制在一定情况下就会表现出来，例如，当系统大受欢迎，有越来越多的客户使用它时；当产品每天要处理的交易数量逐日攀升时；或者，当发现必须保留 6 个月而不是最初设定的一周的数据量时。单独拉伸每个维度，然后综合起来看待，便可暴露出那些隐藏于最初设计中的潜在限制和不足。

架构师可以通过拉伸关键维度，对解决方案进行校核：

- 了解基础设施的规划是否能够应付增长的需求，圈出限制范围。如果基础设施将无法支撑，通过这个操作可以定位隐患，这样就能向应用程序所有者强调；也可在按规划采购基础设施时，对今后的升级路径心中有数。
- 确认在预期的吞吐量下，系统不但能在当天内及时完成全天的任务处理，同时具备峰值储备（head room），才能应对“特别忙碌的日子”，才能在停电后“加班补上”。一个解决方案，如果不能在当天按时完成全部的处理任务，而要依赖周末空闲时间加班加点，那就谈不上有多少未来。

- 对当前的数据访问方案进行校核，确保其在系统伸缩扩展时依然有效。在当前一周数据量下能工作的方案，可能在载入 6 个月的数据量时就无法使用了。
- 确保在系统工作负载上升时，（如果需要）能够以增加硬件和转换处理路径（transition path）的方式进行系统的伸缩扩展。在应用程序部署之前就要完成这种转换，而且它会影响数据的存储和结构。
- 数据量持续上升，导致数据必须在更多的基础设施间进行分割时，要确保应用程序仍然可用。

基于这样的检查，可以发现设计中存在的问题，从而对之重新设计。这时，由于还只是虚拟的设计，技术选型也还没有锁定，业务数据尚未进入库中，要进行重新设计的代价还不太高。

作者简介：

斯蒂芬·琼斯（Stephen Jones）为诸如 Telstra 和 Optus 等澳大利亚公司设计 Tier-1 电信计费系统及相关的高容量处理解决方案，1997 年为美国的 AT&T 公司推出了新的版本。其中的设计工作包括电信计费系统的最初实现，而且重新设计了争议事后付款（post-bill dispute）和防舞弊计费（fraud billing）功能。最近两年多来一直在负责对 Telstra 提供 24/7 的产品支持。

架构师要以自己的编程能力为依托

If You Design It, You Should Be Able to Code It

迈克·布朗（Mike Brown）



在架构上，架构师都期望能够创建出精巧的设计和完美的抽象，来优雅地解决手头的问题。如果在项目中能多安插些新技术，那就更让人有成就感了。但是，最终要实现设计的不是架构师。如果开发人员需要去实现那些架构上的“杂技（acrobatics）”，那将会对整个项目造成巨大影响。

为项目设计架构时，对实现每个设计元素所需的工作量，要做到心中有数；如果以前曾开发过其中某种设计元素，那么估算所需工作量将会容易得多。

不要在设计里使用自己没有亲自实现过的模式，不要使用自己没有用之写过代码的框架，不要使用自己没有亲自配置过的服务器。如果架构依赖于各种你未曾亲身使用过的设计元素，那其中就存有许多负面影响：

- 对开发人员需经历的学习曲线，你将毫无经验。如果不清楚学习某种新技术需要的时间，你也将无法很好预估实现设计所需的时间。
- 对使用这些设计元素时要注意避免的陷阱（pitfall），你将一无所知。真正的实现，肯定不会像那些训练有素的专家所演示的那样完美。如果此前从未接触过这些技术，问题发生时你就钻入死胡同里无从应对了。

- 开发人员的信心将会深受打击。当他们向你询问设计相关的问题时，如果你无法给出确凿的答案，他们将很快对你和你的设计失去信心。
- 将会引入不必要的风险。不了解这些东西而在项目中盲目使用，就会在解决方案的关键元素上打上大大的问号。没人会愿意启动存在不必要的巨大风险的项目。

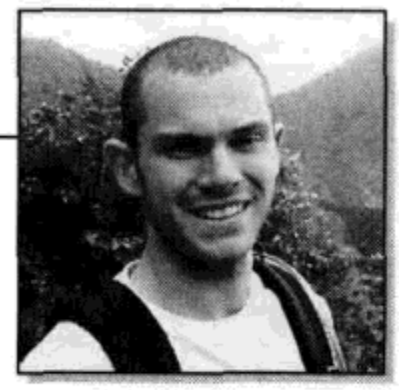
那么，架构师该如何学习新框架、新模式（pattern）和新的服务器平台呢？好，请参看另外一条格言：《架构师首先是开发人员》。

作者简介见第 127 页。

命名要恰如其分

A Rose by Any Other Name Will End Up As a Cabbage

萨姆·加德纳（Sam Gardiner）



我在偶然机会中听到有些人决定在他们的架构中添加更多的层次。开始时他们做得不错，但慢慢地就有所滞后了。他们试图建立的是一个包含业务逻辑的框架。他们一开始不是首先去解决具体的问题，而是希望能有一个框架，可以把数据库包装起来并且能够生成对象。他们设想，这个框架应该使用对象-关系映射（object-relational mapping），应该使用消息机制，还要使用 web service，应该包含各种酷的元素。

不幸的是，他们并没有想清楚这个很酷的框架到底可以做什么，因此不知道该如何称呼它。于是，他们举行了一个小小的起名竞赛，想给框架起个合适的名字。而事实上在此时就应该明白，这其中一定有问题：如果都不知道一个东西应该叫什么，那你肯定不知道它究竟是什么。如果你不知道它究竟是什么，那么你也肯定不能坐下来为它编写代码。

在这个特别的案例里，通过快速浏览整个源代码控制的历史记录，我发现了深层次的问题。毫无悬念，其中有很多的空接口，标明“待实现”！而真正有趣的事情是，接口的名字已经改过很多次，但仍然没有增加任何实质性的代码。开始时，他们把一个接口称为 ClientAPI——事实上，这里的“client”想指的是商业上的客户（customer），不是“客户机-服务器（client-server）”中的“client”——而在最后的版本中，这个接口又被改称为 ClientBusinessObjects。真是个好名字：模糊、不着边际、极易误导他人。

当然，名称无非只是一种指代。一旦相关人都知道，在这里名字只是一个名字，那么便失去了可以赖之前进的设计隐喻（design metaphor）。如果名字根本就是错误的，通过名字根本无法理解其究竟指代的是什么，那么显然很难继续往下做事情，有时甚至根本就无法开始编码。所谓设计，就是要去实现各种意图，例如，快速、廉价、灵活，而名字便是用于承载和传达这些意图的。

如果无法给出合适的命名，那也就无法继续编程。如果发现自己需要多次更改命名，那么最好停下来，直到弄清楚要做的究竟是什么。

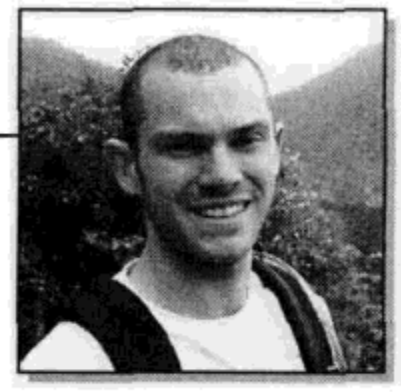
作者简介：

萨姆·加德纳（Sam Gardiner）最初在BBC电脑上使用BASIC语言编写游戏，后来使用过pascal、Mathematica等多种语言，基于LabVIEW处理由原始文本数据文件构成的数据库，因而误打误撞地进入专业软件开发领域。目前，他已经在软件业工作了6年。

稳定的问题才能产生高质量的解决方案

Stable Problems Get High-Quality Solutions

萨姆·加德纳（Sam Gardiner）



现实世界中的编程，并不是要去解决别人给你的问题。在计算机课堂上，你必须要能解决二叉排序（binary-sort）问题。但在现实世界中，最好的架构师不是要去解决难题，而是要围绕难题开展工作。架构师要能够将四处弥漫的软件问题圈起来，并画出其中的各种边界，确保对问题有稳定的（stable）、完整的（self-contained）认识。

架构师必须能从整体上看待杂乱无章的数据、概念、数据和处理逻辑，架构师要能够将它们作为整体看待，并将它们分解为更小的片段或“块（chunk）”。重点在于，这些问题块必须是稳定的，它们在范围上有限且稳定，可以作为系统模块解决。这些问题块应该具备以下特性。

- 内聚性：问题块在概念上是统一的，其中所有的任务、数据和特征都是相关的。
- 能够很好地和其他块分隔开：这些块在概念上进行了规范化处理（normalized）；它们之间很少重叠或根本就不重叠。

就像方向感很好的人不会迷路一样，擅长这方面的架构师，甚至可能不会意识到自己正在这么做。将任务、数据和特征分解开，提供好的系统边缘（edge）或接口，这对他们而言是很自然的事情。注意，我在这里说的接口并非指面向对象语言中的“接口”，而是指系统边界。

举例而言，关系数据库管理系统就有一个很好的系统边界。它能够管理各种类型的数据，把数据序列化为字节流，提供数据组织、检索和提取数据的接口，十分简洁。

有趣的地方在于，如果问题是稳定的，那么，问题解决之后，就永远不会再来烦恼你。在 5～50 年的时间范围内，你也许突然会想在外面加上一个 web，或者可以通过心灵感应（telepathic）来控制的界面，但是核心系统不需要为此做出任何改变。因为问题是长期稳定的，所以这个核心系统也是长期稳定的。

当然，代码需要写得相当整洁。如果问题被处理得很干净，在没有什么例外的情况下，代码当然可以写得很整洁。整洁的代码有很多好处，如易于测试、便于审查，这也就意味着实现质量可以非常高。在没有搞乱代码的前提下，架构师就可以集中精力处理那些业务域之外的事情了，如使用可靠的消息通信机制、使用多线程来提升性能，甚至使用像汇编代码这样的低级语言等，这些对用户而言是不可见的。问题没有改变，你因此可以集中精力来提升功能的质量。

只要问题是稳定的，你就可以创建出一个拥有稳定设计（a stable design）的系统。稳定的设计可以让你集中精力，打造出高质量的应用程序。

作者简介见第 153 页。

天道酬勤

It Takes Diligence

布赖恩·哈特（Brian Hart）



架构师常常被描述为一种注重创造力与问题解决能力的职业。具有创造力诚然是成功架构师的一项重要特质，不过，成功架构师还须具备另一项同样重要的特质——勤奋（diligence）。勤奋体现在很多方面，但归根结底是指具备坚强的毅力，并且对系统的每项任务和每个架构目标，都能投入足够的精力。

勤奋经常和平凡（mundane）携手同行。成功的架构实践，在很多方面都是简单易行平凡无奇的。高效的架构师，通常能使用极为普通的每日和每周检查列表（checklist），提醒自己去处理那些已经想明白但实践起来还不够熟练的任务。如果没有这些不起眼的检查列表的提醒，架构师很快就会在时间上处于被动状态，无法获得可量化的进展。因为少了日常的勤奋，架构设计可能就会在不知不觉间违背了各种已知的理论原则。很多时候，不是能力不足导致项目的失败，而是由于勤奋不够，紧迫感不强。认识到这点很重要，许多失败项目的反思回顾（retrospective）也都揭示了这点。

勤奋还意味着要求架构师必须真正做好那些看似简单的任务，坚守承诺。这些承诺和相当宽广范围内的约束与期望相关，其具体表现形式多种多样。比如：

- 坦然面对客户在预算和时间上给出的约束。
- 要做能提升架构师效率的工作，而不仅仅只是做自己喜欢的工作。
- 坚持贯彻使用的过程和方法学。
- 承担责任。

阿图尔·加万德（Atul Gawande）（译注 1）在他了不起的著作《精进不止：手术室中的沉思》（Metropolitan Books 出版社出版）中，这样评述勤奋在医学界的意义：

> 医学上的真正成功来之不易。它需要意志力和创造力，也要求从业者关注细节。但我在印度所得的经验是，任何领域、任何个人都有可能取得成功。我无法想出还有哪里的环境比之更恶劣了。但在那里依然能够诞生惊人的成就。……在那里，我所发现的是，我们依然可以精进不止。成功不需要极高的天分，它需要的是勤奋，需要的是清晰的道德观念，需要的是创造力。最重要的是，你必须怀有不断尝试的信念。

作者简介：

布赖恩·哈特是 CGI 公司的一名执行顾问，CGI 公司是业内领先的 IT 与业务流程服务提供商。布赖恩主要参与国家和政府部门的 J2EE 程序架构和设计。他于 1997 年开始进入软件行业。

译注 1：阿图尔·加万德（Atul Gawande），毕业于美国哈佛大学医学院，印度裔外科医生，现为哈佛大学教授。著有畅销书《Complications》和《Better》。台湾天下文化出版社曾翻译引进，译者廖月娟，译名分别为《一位外科医师的修炼》和《开刀房的沉思——一位外科医师的精进》。在《Better》一书的《跋》中，加万德写道，他不甘心一辈子成为美国医疗机器中的一个小齿轮，少了他一个也无所谓，他要成为“异数”，于是努力寻找成为“异数”的五个努力方向，分别是：随口问问，别猛吐苦水，找目标计数，勤于写作，勇于改变。

对决策负责

Take Responsibility for Your Decisions

周异（Yi Zhou）



由于软件架构师比组织中的其他人更有影响力，所以他们必须对自己做出的决策负责。研究表明，超过三分之二的软件项目要么彻底失败了，要么未能成功交付（表现为项目延期、预算超支、客户不满意）。究其根本原因，有许多是由于软件架构师做出了不当的决策，或者无法最终执行正确的架构决策。

怎样才能成为负责任的软件架构师，做出有效的架构决策呢？

首先，无论是以敏捷的形式还是传统的形式做出架构决策，都必须对决策过程有充分的认识。只有满足以下两个条件，才算已经完成架构决策：

- 该决策已以书面形式记录下来；由于架构决策很少是无关紧要的小事。它们必须经过校核证实，并可被追溯（traceable）。
- 必须已和执行该决策及会直接或间接受其影响的人进行过沟通，达成共识。

第二，要定期对架构决策进行复审；对照检查决策的实际效果和预期结果；识别出哪些已被证明为有效决策，哪些被证明为无效决策。

第三，要贯彻架构决策。有许多软件项目，软件架构师仅参与设计阶段，随后即转向其他项目，或者咨询合同到设计阶段完成就终止了。架构师怎样确保经过其深思熟虑的架构决策最终会被正确实现呢？他们只有全程跟进实施过程，才能够确保最到位地实现其设计意图。

最后，可以将一些决策委托给相应问题域的专家。许多架构师错误地认为，每个架构决策都必须出自其自身。因此，他们将自己定位为“无所不知（know-it-all）”的专家。在现实中，并没有所谓的全能技术天才（a universal technical genius）。对有些领域，架构师相当纯熟；对有些领域，他们有所了解；对有些领域，他们则根本说不上话。对于不精通的领域，老练的架构师会将决策委托给合适的专家。

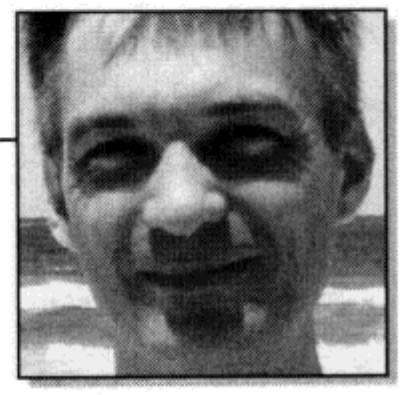
作者简介：

周异（Yi Zhou）目前是一家著名生物技术公司的首席软件架构师，专长于为医疗设备和个性化疾病管理设计软件平台。在软件开发生命周期的各个方面，他有近 20 年的经验，擅长于结合业务和技术进行战略规划、过程改进、架构和框架设计、团队建设和管理及咨询。

弃聪明，求质朴

Don't Be Clever

埃本·休伊特（Eben Hewitt）



智力超群、足智多谋、深思熟虑、知识广度与深度兼备、追求准确性，这对任何人来说都是值得称道的品质。对架构师来说，这些素质尤其被看重。

然而，聪明（Cleverness）也包涵某些额外的隐含意义。它也暗指能快速想出脱身之计的能力，但这种本领最终要依靠一些小伎俩（gimmick）、骗局（shell game）或调包计（switcharoo）。从中学时代起，我们就应该已经领教过那些巧舌如簧的聪明家伙，他们总是靠玩文字游戏或逻辑谬论来取胜。

聪明的软件价格昂贵，不易维护，僵脆易折。所以，不要追求聪明，尽量用最浅显易懂的质朴（dumb）方法，恰如其分地进行设计。恰如其分的设计舍弃聪明。如果聪明看似必不可少，那么只能说明问题仍未正确界定。这时要去重新界定问题，直到一切再次变得浅显质朴为止。你可以使用粗粉笔画些草图，保持常规思维。别去理会什么流行风潮。只有真正睿智的架构师才懂得如何保持质朴。

小聪明会诱导我们在软件开发中使用奇技淫巧。不要依赖某个技术细节使软件跑起来。我们不是鲁布·戈德堡（Rube Goldberg）（译注 1），不是麦吉弗

译注 1：鲁布·戈德堡（全名 Reuben Garret Lucius Goldberg），是美国著名的漫画家、雕刻家、作家、工程师、发明家，全美漫画家协会的创立者和主席。1948 年因他的政治漫画而获得普利策奖。因创作鲁布·戈德堡机械（Rube Goldberg devices/machines）系列漫画受到大众欢迎。他的漫画系列《发明家》异想天开地表达用复杂机械完成简单任务的可笑设计，想不到因此而闻名于世。他画了许多用极其复杂的方法从事简单小事的漫画，赢得了许多读者的喜爱。比如把鸡蛋放进小碟子这种事，在戈德堡笔下大概是这样的：一个人从厨房桌子上拿起晨报，于是牵动了一条打开鸟笼的线，鸟被放出来，顺着鸟食走向一个平台。鸟从平台摔到一个水罐上，水罐翻倒，拉动扳机，使手枪开火。猴子被枪声吓得把头撞在系有剃刀的杯子上，剃刀切入鸡蛋，打开鸡壳，使鸡蛋落入小碟子中。如今，“Rube Goldberg”已成为“简单事情复杂化”的代名词。

（MacGyver）（译注 2），无法随时用一只回形针、一支爆竹或一片口香糖就能从帽子里揪出各种复杂的设计来。先清空你的大脑，不要带着闭包、泛化及如何操控堆中的对象这些丰富的知识去解决问题。当然，有时候我们的确刚好需要这些玩意，但是这种情况往往比我们以为的要少。

越来越多的开发人员已经懂得如何实现和维护质朴的解决方案（dumb solutions）。在质朴的方案中，每个组件只做一件事。这些组件耗时少，易于创建，以后要改变也无须花很多时间。它们可以从当前使用的构造块（building blocks）中继承已有的优化（optimization）。这些组件会随着鲜活的设计开发过程自己涌现（emerge）出来，那时你可以感觉到它们的优雅和简洁。聪明的设计僵硬难改，其细节会对全局产生太多的牵扯，往往会一触即毁。

作者简介：

埃本·休伊特（Eben Hewitt）是某家资产数十亿的零售公司的软件架构团队负责人，目前主要负责设计和实现该公司面向服务的架构（SOA）。其作品《Java SOA Cookbook》即将由 O'Reilly 出版。

译注 2：麦吉弗（MacGyver）是 20 世纪 80 年代在美国热播的冒险题材连续剧《MacGyver》（中文译名《百战天龙》）的主人公，麦吉弗是个足智多谋的人，擅长用普通生活用品作为工具帮助自己和搭档摆脱困境，常常是一把小刀走天下。他拥有广博的知识，还有一套特立独行的“麦吉弗主义”，凭着过人的智慧，化解种种危机。

精心选择有效技术，绝不轻易抛弃

Choose Your Weapons Carefully, Relinquish Them Reluctantly

查德·拉·瓦因（Chad La Vigne）



作为软件设计开发的老手，每个架构师通常都有一些让自己屡屡取胜的武器用来武装自己。这些技术由于种种原因深受架构师青睐，排在其首选解决方案列表的前几位。其中大多数是因为在激烈的竞争中胜出而在军械库中获得了合法席位。尽管如此，它们的地位还是会接二连三地遭受到新技术的威胁。我们常常会被迫放下曾经选择的武器而改用那些新家伙，但是，最好还是不要过快放弃那些你可以信赖的武器。而改用那些未经考验的试用品，这要冒很大的风险。

这并不是说，某种技术一旦已经入围首选列表（list of favorites），就可以永留罔替，更不是指你可以像鸵鸟一般将头埋进沙子中，对软件开发技术的发展进步埋头不理。每种技术都有各自的生命周期，都终将会被后来者取代。技术飞速更新，优秀的解决方案层出不穷。作为架构师，我们应当紧跟产业趋势，但我们并不必急于去做拥抱那些羽翼未丰的新技术的先行者。通常，为新技术做第一个吃螃蟹的人，并不能享受到多大的好处，反而可能会遭遇不少挫折。

选择新技术虽然有风险，但其价值在于能为你带来质的飞跃。很多新技术都打着这样的口号登场，但真正能兑现承诺的实际上很少。只看见新技术中的技术优势非常容易，但要将这些技术优势推销到利益相关者手里往往很难。在决定使用新技术来另辟蹊径之前，问问自己，这样的决定能给实际业务带来什么好处。如果从商业角度考虑的最好结果也只是“根本没有人会在意这些”，那么一定要重新考虑你的决定。

另一点也不容忽视，那就是新技术的缺陷可能带来的成本问题。这些成本可能会很高而且难以统计。使用熟悉的技术时，你对它的各种特质了如指掌。无视新技术中包含的各种缺陷，是非常幼稚的。往项目中增加自己未曾解决过的额外问题，将会破坏之前所做的成本预估。而用熟悉的技术来实现解决方案时，对所需成本更能做到心中有数。

最后要考虑的一点是技术的未来前景。如果真能简单到只管选择最优秀的技术那倒也确实不错，但事实往往并没那么简单。技术优秀并不意味着总能胜出。过早预测胜出者，无异于是在进行一场回报不明的赌博。等新技术鼓噪的声势过去后，再看看它们是否能生存下来，成为确实实用的武器。也许那时你会发现早前很多所谓的新技术都已经销声匿迹不知所终了。不要为了那些没有未来的新技术，把项目置于险境。

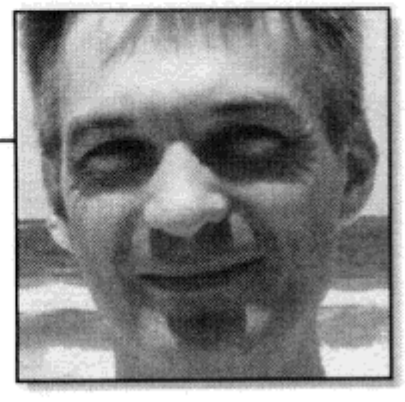
软件架构师工作很大的一部分，是要选择用以攻克难题的合适技术。精心选择熟悉的武器，不到万不得已绝不轻易抛弃它们。这些技术在过去给你带来了成功，尽量让它们在未来也能为你带来胜利，同时，以审慎的态度更新你的技术武器库。

作者简介见第 111 页。

客户的客户才是你的客户！

Your Customer Is Not Your Customer

埃本·休伊特（Eben Hewitt）



在软件设计的需求会议上，要这样想象：你的客户并不是你的客户。这实际上真的不难做到，因为，事实便是如此。

没错，你的客户确实不是你的客户。你的客户的客户，才是你的客户。如果你的客户的客户赢了，你的客户也就赢了。这意味着，你也赢了。

假设你正在编写一个电子商务应用程序，那么该仔细考虑的应当是那些会在那个网站上购物的人的需求。他们需要安全的信息传输，需要数据加密存储。而这些需求，你的客户可能不会提及。如果发现你的客户疏忽了他们的客户的需求，你要指出来，并说明原因。

如果你的客户有意无意地忽视他们的客户所看重的重要事项——这种情况常有发生——那么，请考虑放弃这个项目。如果你的客户 Sally 不想每年为 SSL 掏腰包，而且打算把信用卡数据以纯文本格式存储（因为这样做更省钱），这样的需求可不能轻易就答应了。明知是糟糕的主意，但你如果竟然也同意接手，这无异于在谋杀客户的客户。

需求收集会议不是项目实现讨论会议。除非所涉问题非常明确或大家都已理解，不要让客户使用与具体实现相关的术语。允许客户发表他们或显不切实际过于理想化的理念（Platonic ideal），描述他们的概念及目标，但不要听任他们自以为是地给出解决方案，他们有时居然还会使用技术术语。

这看起来似乎难以做到。那么，该如何在这类会议上坚持这样的原则呢？只要牢牢关注于你的客户的客户的需求即可。记住，尽管给你开支票的是你的客户，但你应始终清楚一点：自己应该诚实遵循那些最佳实践所揭示的规律，这样你才能产出客户真正需要的东西，而不只是他们声称自己需要的东西。当然，你需要为此进行大量讨论，在讨论中要准确清晰地陈述你的方案及理由。

如同生活中的许多事情一样，有一首诗或许对此做出了最好的诠释。在 1649 年，理查德·洛夫莱斯（Richard Lovelace）（译注 1）在《出征前致露卡斯塔》的尾句中写道："我又怎配真的爱你，亲爱的/如果我不更爱自己的荣誉（I could not love thee, dear, so much,/Loved I not honor more.）。"

我们也不配称得上真正关爱我们的客户，如果不能更为关爱他们的客户。

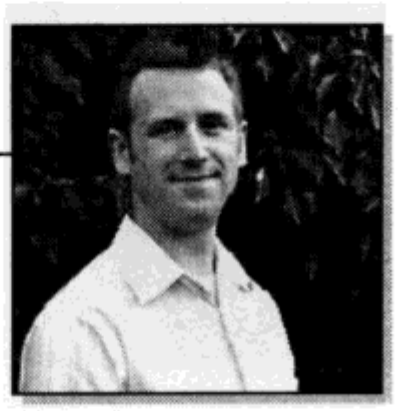
作者简介见第 161 页。

译注 1：理查德·洛夫莱斯（Richard Lovelace，1618~1658 年），是 17 世纪英国查尔斯一世宫廷里的骑士派诗人之一，出生于英国肯特郡一个富有之家。《出征前致露卡丝塔》是他在奔赴战场前写给未婚妻露卡斯塔的告别诗。他最著名的诗是《在监狱致阿尔泰亚》，其中有这样的诗句："监狱不是石墙砌成的，笼子也不是铁栅制成的。"1642 年，在英国"大叛乱"时期，他曾因保皇主张被短期逮捕入狱，诗歌《在监狱致阿尔泰亚》正是创作于此期。1646 年他在法国军队服役，回国后再次因参与保皇派活动而入狱。他死的时候一贫如洗。他的兄弟将他的大部分诗歌汇集为《露卡丝塔》（Lucasta: Posthume Poems），于 1659 年出版。

事物发展总会出人意料

It Will Never Look Like That

彼得·吉拉德莫斯（Peter Gillard-Moss）



事物发展总会出人意料。在设计时，人们很容易陷入误区，投入太多的时间在设计上。而且对实现会和设计完全一致，抱有过高的信心。所谓的详细设计，极易产生误导，使你以为在设计中已经覆盖了方方面面。设计越详细，研究越深入，你的信心就越足。但这确是一种错觉，因为事物发展总会出人意料。

事实是，无论研究得多么深入透彻，无论设计是如何深思熟虑，事情最后总会变得和你想的不一样。总有些事情会冒出来，有些外部因素也许就会影响到设计：不正确的信息，某种限制，某人代码中的古怪行为等。或者你也可能会发现自己犯了一些错误：某处疏忽了，某处假设错了，遗漏了某个微妙的概念。或者是由于有些事物也发生了变化：需求、技术，或者是有人可能找到了更好的办法。

设计中的这些微小变化累积起来，很快就需要对设计进行一次较大的变更。不久，就发现最初的概念已经支离破碎了，你又不得不站回到图板前重新思考设计。之后，你做出了更多详细的设计。下一个版本的架构图景显得更加清晰，更加激进，也更加完美了。

但好景不长，不久同样的事情又再次发生。变化再次出现，设计开始出现偏移，开发人员越是试图尽量维持住已经走样的设计，就越会破坏原来的设计。你最终只能高声大喊“这样做当然会有错误；因为从来就没设计成要这样做!”

设计是一个不断发现的过程。在实现时，我们会发现通常无法预知的新信息。设计是在不断变化的世界中持续进行探索试验的过程，只有接受这点，我们才能明白，设计过程也必须保持灵活性和连续性。固守最初的设计，并迫使实现遵循最初的设计，只能产生更为糟糕的结果。因此，要理解一点，事物发展总会出人意料。

作者简介：

彼得·吉拉德莫斯（Peter Gillard-Moss）供职于 ThoughtWorks 公司，是泛文化基因进化论者（general memeologist（译注 1）），住在英国。自 2000 年开始，他开始在 IT 界工作，做过许多项目，从面向公众的媒体网站和电子商务网站，到富客户端的银行应用软件和局域网内的企业应用。

译注 1：memeologist ，memetic evolutionist 的合成词，文化基因进化论者。

选择彼此间可协调工作的框架

Choose Frameworks That Play Well with Others

埃里克·霍索恩（Eric Hawthorne）



软件框架是系统的基础，在选择时，不仅要考虑每个框架自身的质量和特性，也要关注共同构成系统的各个框架之间是否能和谐共处，另外还要关注，随着系统不断演化，是否能方便地向其中加入新的软件框架。即，必须选择彼此之间没有重叠，而且开放、简洁、精专（specialized）的框架。

每个框架或第三方库（third-party library），如果既能专注解决某个独立的逻辑域或关注面（concern），又不会侵入其他必需框架的领域或关注面，那是最好不过了。

如果框架间互有重叠，则要确保了解候选框架在逻辑域或关注面上的重叠程度。必要时，可以绘制韦恩图（Venn diagram）（译注 1）来辅助。领域内存在大幅重叠的数据模型，或者以略微不同的方式解决非常相似的关注面的两个实现，都会带来不必要的复杂性：概念或表示上的细微差别必须用杂碎的胶水代码（glue code）打补丁或是进行映射。到最后，你有可能不仅引入了复杂的胶水代码，而且，所得的也只能是两个框架核心功能的最小公分母。

为了尽量减少框架间互有重叠的概率，要根据系统需求的应用场合，选择精专强大（high utility-to-baggage ratio）的框架。强大有用的框架，能够提供项目所需的功能或数据表示法。而“无所不能”型的框架，依靠的是囊括一切、独霸

译注 1：韦恩图（Venn diagram），用于描述集合间的关系及其运算，直观、形象、信息量大且富有启发性。一般用矩形表示全集 U，用圆表示 U 的子集 A，B，C 等。

天下的理念。框架是否强制数据的表示（data representation）和控制（control）混在一起？是否其数据模型或包和类的集合，远远超出系统实际所需？是否要求必须成为该框架的原教旨主义者（fundamentalist），限制选择其他适用框架？往框架中混入（mix）其他东西,是否会超出可控复杂性的极限？如果要选用一个“无所不能”型的框架，那它最好能够提供项目所需全部功能价值 75%以上。

系统应该由多个相互独立的框架组成，其中每个框架都精专于各自的领域，但同时又非常简洁（simple）、包容（humble）和灵活（flexible）（译注 2）。

作者简介：

埃里克·霍索恩（Eric Hawthorne）自 1988 年起，专门从事面向对象和分布式系统的架构设计和开发。他最初在 Macdonald Dettwiler（一家加拿大的系统工程公司）工作了 10 年，期间除了其他收获外，有幸从菲利普·克鲁采恩（Philippe Kruchten）（译注 3）那里学到了不少架构技术。

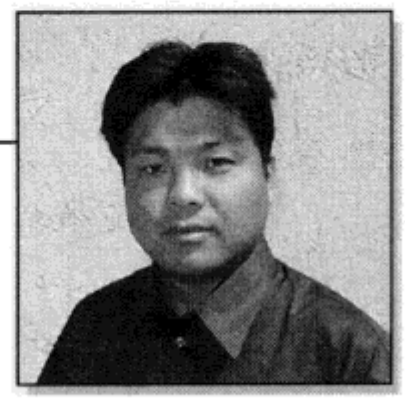
译注 2：原文作者在结尾使用 simple、humble 和 flexible 三个单词，呈现押韵的效果，简洁优雅。

译注 3：菲利普·克鲁采恩（Philippe Kruchten）是名加拿大软件工程师，是温哥华不列颠哥伦比亚（UBC）大学的软件工程教授，Rational 软件公司的 RUP 总监，4+1 视图模型的发明者。

着重强调项目的商业价值

Make a Strong Business Case

周异（Yi Zhou）



作为软件架构师，你是否曾在为架构项目争取资金时遭遇到困难？对架构师而言，软件架构的好处显而易见，但对许多利益相关者（stakeholder）而言，软件架构却显得虚无缥缈（mythical）。大众心理学表明，大部分人会相信亲眼所见的东西（seeing is believing）。但是在项目早期，我们无法向利益相关者充分证明优秀软件架构带来的实在价值。如果他们不是软件业从业人员，那就难上加难了，因为利益相关人士大多缺乏软件工程方面的知识。

大众心理学还表明，大部分人对能感知的东西即认为是真实的现实（perception is reality）。因此，如果能够控制人们对你的架构提案的认知方式，无疑也将能影响他们作出的反应。如何掌握利益相关者的认知呢？着力强调项目的商业价值！那些可以赞助你的想法、拥有预算决定权的人，几乎都是受商业价值驱使的。

我在职业生涯中，曾经多次使用下面五步，成功将架构提案打造为典型的商业项目：

1. 形成价值陈述（value proposition）。价值陈述是你的决策摘要（executive summary），用以说明组织的业务为何要采用某种特定的软件架构。这一步的关键，是要将你的架构方案与其他既有方案或可选方案进行比较。重点应放在说明其在提高生产力、改进业务效率方面的能力，而不是强调其采用的技术如何高明。

2. 建立量化的度量标准（metric）。对于承诺要交付的价值，需要在合理范围内进行量化。量化得越具体越到位，项目也将越具说服力，越能让人相信好的架构可以带来丰厚回报。越早建立度量标准，就越容易把握人们的认知，帮助推销自己的架构提案。

3. 回过头来关联传统商业的衡量方式。如果能将技术分析转化为财务数据，则会更为完美。毕竟，传统商业衡量方式中唯一不变的参数便是经济收益。如果不喜欢做财务性的工作，可以找些商业分析师来做你的搭档。

4. 知道该在哪里停止。在知道该在哪里停止之前，要准备好一张路线图（roadmap）用以捕获远景目标，清楚地知道每一个里程碑将带来的商业价值。让利益相关者自己决定在何处停止。如果每处的商业价值都十分显著，那么，很可能你会获得持续不断的资金支持。

5. 寻找恰当的时机。即使按照前面四个步骤创建了稳固的商业项目提案，但如果时机不对，可能仍然无法成功推销你的点子。我记得自己就曾有一个提案迟迟无法获批，直到另一个项目因为糟糕的架构设计以彻底失败告终时，我的提案才获得通过。所以，还要明智选择恰当的时机。

作者简介见第 159 页。

不仅仅只控制代码，也要控制数据

Control the Data, Not Just the Code

查德·拉·瓦因（Chad La Vigne）



源代码控制和持续集成，是管理应用程序构建过程和部署过程的优秀工具。数据方案（schema）和数据内容通常会随着源代码的变化而变化，它们也是这一过程的重要组成部分，因此也需要对之进行类似的控制。如果构建和部署过程的详细步骤列表中包含要进行数据更新的操作时，一定要明白这点。你一定碰到过这样的过程列表，看起来大致像是下面这样的：

1. 创建脚本列表，这些脚本要按照一定次序运行。
2. 把这些脚本通过 E-mail 发送给数据库管理员。
3. 数据库管理员把这些脚本复制到某个地方，通过定时任务调度（cron job）来执行。
4. 检查脚本的执行日志，祈祷所有的脚本都能够运行成功。要是必须重新运行这些脚本，真不知道到底会发生什么事情。
5. 运行校验脚本，进行数据抽样检查。
6. 对应用程序进行回归测试（regression test），看哪些地方被破坏了。
7. 编写脚本，补上丢失的数据，修补被破坏的地方。
8. 重复上面的操作。

好吧，说得也许有点夸张，但也应该八九不离十。为了能够成功完成数据库迁移，很多项目都要经历这种像玩杂技一样的工作流程。

由于某些原因，在架构规划过程中，数据迁移部分似乎很容易被架构师忽略。最终，数据迁移往往只是作为一项事后补救措施，而且整个过程由手工操作完成，相当脆弱（brittle）。

这样一种复杂的网状工作过程，失败的几率很高。更糟糕的是，由于数据方案和数据内容变化造成的错误，并非总能被单元测试捕获到，因此通过夜间构建报告（nightly build report）也无法发现错误状况。这种错误经常会在迁移已经完成后的构建中，突然冒出头来，闹得人心惶惶，十分烦人。要消除这些数据库问题，只有依靠单调乏味的手工回退。而且验证解决方法的有效性，往往也会变得更加困难。完全自动化构建过程（a completely automated build process）的价值，在于它能将数据库恢复到某个已知状态下。如果一个问题很明显是因数据库迁移所引起，使用这种方法来修复，完全自动化构建过程的价值就体现得更加明显。删除数据库并迅速将数据库重新恢复到与应用程序的某个特定版本相兼容的状态，如果不能做到这点，也同样无法做到可在不同代码版本间迅速地切换。

数据库的变化不应该影响构建活动的连续性。要把数据库也作为一个构建单元包含在内，做到一次性构建整个应用程序。对数据方案和数据内容的管理，应当尽早无缝集成到自动化的构建和测试过程中，还必须提供回退功能；这样做能获得巨大的收益。做得好，可以省去在夜间构建失败后顶着巨大压力去解决问题的痛苦时光。最起码，如果继续开发需要重构数据访问层，大家丝毫也不必担心。

作者简介见第 111 页。

偿还技术债务

Pay Down Your Technical Debt

伯克哈特·赫夫纳盖尔（Burkhardt Hufnagel）



对任何已投入实用的项目（也就是说，有客户在使用其产品），无论是要修复缺陷，还是要添加新功能，总有必须修改产品的时候。在那点上，会面临两个可能选择：花合适时间“一次做对”，或者取巧走“捷径（shortcut）”，争取尽快推出修改后的产品。

一般来说，业务相关人员（销售/市场部门和客户）希望这些修改越快越好，而开发人员和测试人员更期望能够花合适时间进行设计、实现和测试后，再交付给客户。

作为项目的架构师，必须决定怎样做更有意义，并说服决策者采纳你的意见；另外，和大多数架构设计问题一样，要对此进行权衡。如果相信系统足够稳定，那么走“脏但快速（quick and dirty）”的路线也许合适些,能将修改快速纳入产品中。这很好，但是要知道，这样做，项目将招致一些后面必须偿还的“技术债务（technical debt）”。这里所说的“偿还（repayment）”，是指假设能够回到刚开始的时候，而当时的时间和资源又都充足，以在那样的条件环境下会采取的方式重新进行修改。

那么，为什么修改的时机还有这么多讲究呢？这是由于“脏但快速”的修复有隐性成本（hidden cost）。金融债务的隐性成本是所谓的“利息（interest）”，大多数使用信用卡的人都知道，信用卡债务的利息是很昂贵的。对于技术债务，它

的利息表现为系统的不稳定性，以及由于临时性手段和缺乏合适的设计、文档工作和测试带来的不断攀升的维护成本。而且，和金融债务一样，在本金完全还清之前，还必须定期偿还利息。

现在，对技术债务的实际成本已有了更多了解，你可能会觉得成本过高，负担不起。但是，当面临选择究竟是让开发人员尽快完成修复，还是承受严重冲击时，通常会选择“快点修复掉吧”。所以，不妨承受些损失，先让修改尽快（ASAP，As Soon As Possible）加到产品中。但是，可千万不能就此止步。

一旦修复完毕，记得安排开发人员回头去妥善修复解决，纳入下一个计划发布的版本中。这等同于在月底给信用卡还款，还清债务维持账户平衡，这样就不必付息了。这种方式，不但满足业务所需的快速变化，也可让项目免于陷入债务牢笼之中。

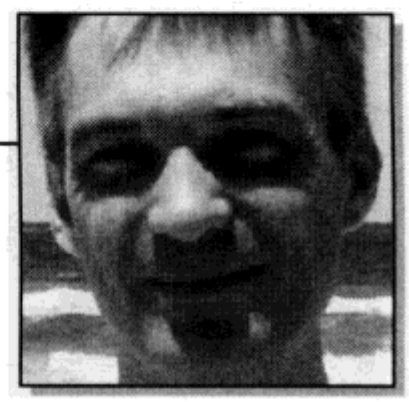
作者简介：

伯克哈特·赫夫纳盖尔（Burkhardt Hufnagel）自 1978 年开始就一直致力于创建最佳用户体验，他是 LexisNexis 公司的杰出软件架构师。

不要急于求解

Don't Be a Problem Solver

埃本·休伊特（Eben Hewitt）



架构师多半是开发人员出身。开发人员的主要职责是解决编程问题。相比架构问题，编程问题的范围更狭窄。很多编程问题是很小但比较棘手的算法问题。这些问题在编程面试、书本及大学课程上经常以很抽象的形式进行描述，仿佛存在于虚空中，和现实世界毫无关联。但复杂棘手的问题又是那么诱人。久而久之，我们便不能自已地喜欢上来自这些问题的挑战。我们不会过问这些问题是否有意义、是否有趣、是否有用或是否符合伦理道德。也没有人会因为我们思考这些问题和更广领域之间的关联而付报酬。越是难以解决的难题，越会吸引我们一心想着去解决。我们沉醉于解决编程面试题，这些题目通常都事先不切实际地设定好一些约束。于是习惯了不去质疑约束。但是，这些难题只是为了引导我们去发现老师、面试官或导师已经知道的东西而设的一些教学工具而已。

架构师和开发人员因此学会了迅速进入问题解决模式（problem-solving mode）。但是有时候，没有解决方案才是最好的解决方案（the best solution is no solution）。有许多软件问题根本就不需要解决。它们之所以看似问题，是因为我们只关注它们的表面症状（symptoms）。

来看内存托管（managed memory）的问题。在内存托管平台上开发的程序员，无须去解决内存问题，即使真有这样的要求，他们中的不少人估计也无法解决。这些平台对内存管理的解决方案，可以部分描述为：不存在内存管理问题，所以也就无须开发人员去管理内存的问题。

再来看构建过程。复杂的构建过程要求使用大量相互关联的脚本，而且这些脚本要遵循很多标准和规范。你可以运用自己高明的脚本技能和最佳实践去解

决这个问题，这确实也非常过瘾。这样做，你无疑也会给同事们留下深刻印象。确实，没有人能够不解决这些问题也给人留下深刻印象。然而，如果能后退一步，看清我们正在解决的不是构建问题，而是自动化（automation）和可移植性（portability）的问题，或许我们就已经转向使用一些现成工具，它们早就抽象了这些问题并很好地解决了。

由于架构师往往习惯于迅速进入问题解决模式，我们常常忘记，或者根本就不知道，该如何去审视（interrogate）问题本身。我们应该学会像长焦镜头一样，不断地拉近放远，以确保正确的锁定问题，而不是只一味地接受别人给出的问题。在需求面前，我们不应该成为被动的储存罐，像糖果盒一样时刻准备着掏出各色各样的聪明点子。

不要立即着手去解决摆在面前的问题，而要看看自己是否可以改变问题。问问自己，如果没有这个问题，系统架构会是怎样的？这样的自问自答，可能最终帮你找到更加优雅可行的解决方案。有时业务问题确实需要得到解决，但有时，或许并非那么迫在眉睫。

我们必须戒除“问题解决迷恋症”。我们往往喜欢接手问题，仿佛自己是一名密探，在欧洲的某座桥上，手里是刚收到的、藏着最新使命的自毁信封。在立即去寻找问题的答案之前，还是先想想，如果根本就不存在这个问题，这个世界又将会怎样。

作者简介见第 161 页。

打造上手（Zuhanden）的系统

Build Systems to Be Zuhanden

基思·布雷思韦特（Keith Braithwaite）



我们是工具制造者。我们制造的系统，一定要能够帮助人们——通常是其他人——做事，否则就失去存在的意义，我们也将无法从中获得报酬。

马丁·海德格尔（Martin Heidegger）（译注 1），是 20 世纪具有影响力的德国哲学家，他对人们在生活中体验工具（或者更一般地讲“设备”）的方式进行了深入研究。人们使用工具来达到目标，工具只不过是达到目的的一种手段。

好的工具会让用户感到很“上手（zuhanden）”，具有“轻巧便利”的特性。这样的工具，能被用户直接体验；使用时用户可以不假思考，无需理论思考（theorisation）。我们很容易掌握这样的工具，并用其来达到目标。在使用时，它消失（vanish）了！工具变成了用户身体的延伸（extension），中间的边界消失不见了。“上手”工具的一个标志，便是它会变得无形无迹（invisible）、无法感知（unfelt）、微不足道（insignificant）。

想象一下，使用锤子钉钉子或用钢笔书写时是什么样的感觉。体会其中的直接性（immediacy），回味那种工具仿佛是身体的无缝扩展（a seamless extension）的感觉。

译注 1：马丁·海德格尔 [Martin Heidegger 1889.09.26－1976.05.26]，德国哲学家，20 世纪存在主义哲学的创始人和主要代表之一。在其主要著作《存在与时间》中，提到了“上手状态”（Zuhanden，ready-to-hand）和“当前在手状态”（Vahanden，present-at-hand）。“上手状态”描述的是主客体浑然一体的状态；“当前在手状态”指的是主客体分离、产生对立的状态。《存在与时间》奠定了海德格尔一生哲学活动的基础，被视为现代存在主义哲学的重要著作。

与之相反，通常当工具出错时，用户可能会体验到“vorhanden（当前在手状态）”的工具。工具与目标隔离开来，横跨在用户和目标之间，要求用户留心注意。这时，工具成为了需要进行研究的对象。用户不能再继续朝向目标前进，必须首先对工具进行处理，否则将无法继续迈向目标。对技术人员而言，我们为用户构建系统，当屡次收到缺陷报告时，便会体验到系统处于“vorhanden”状态。对我们而言，这时工具就成为需进行询问（of enquiry）、理论化（of theorising）和进行调查（of investigation）的对象。工具成为了研究的对象。

然而，对我们构建的工具，用户体验应该是“上手的（zuhanden）”。这是成功的决定性因素。在用户使用时，系统架构是否无形无迹？用户界面是否方便自然？还是，系统要求用户使用时特别小心留意，分散了他们达成目标的注意力？

作者简介见第 21 页。

找到并留住富有激情的问题解决者

Find and Retain Passionate Problem Solvers

查德·拉·瓦因（Chad La Vigne）



组建一支汇聚优秀开发人员的团队，是确保软件项目成功非常重要的事情之一。保持团队长期稳定，尽管不像说起来那么容易，却十分重要。因此，要精心筹建优秀的开发团队，一旦建成，务必竭力维护。

大多数人可能都会同意，要找到一流的开发人员，需要全面严格的技术面试。但究竟如何算是“全面严格”？并非要候选者回答出与极为隐晦的技术细节相关的难题。通过具体的技术知识筛选，无疑是遴选过程的一部分，但是把面试变成认证测验，并不能确保获得真正的人才。我们所要找的，是那种具备解决问题的能力和激情的开发人员。工作中使用的工具肯定会改变，真正所需的，是那种无论技术如何变化，都善于攻克各种难题的人才。即便能背出某个 API 的全部方法，也不能说明此人具备解决问题的才能和激情。

但是，通过要求候选人解释其诊断性能问题的方法，便可深入了解其解决问题的方法。如果想了解开发人员吸取经验教训的能力，则可以询问，如果有机会让他将最近的项目重新再做一次，将会对哪些地方进行调整。好的开发人员对工作充满激情。询问他们过去的经验，就可以见识到他们的激情，洞察到单纯询问琐碎技术问题所无法提供的信息。

如果你已打造了一支强大的团队，定会在权力范围之内竭尽所能维持团队稳定。虽然对于像补偿（compensation）之类的人员存留因素（retention factor），你可能鞭长莫及，但要确保在一些细微之处留意用心，这将有助于营造健康的工作环境。好的开发人员，常常能从认可（recognition）中获得强烈的激励。要充分发挥个中优势，对杰出表现予以通报表扬。找到优秀的开发人员难度颇大；但让人们知道他们的价值并非难事。千万不要错过提高士气和生产力的绝佳机会。

提防批评过度（negative reinforcement）。批评过度，可能会扼杀开发人员的创造力，降低其生产力。更糟糕的是，可能会在团队内部造成纷争。好的开发人员都非常聪明；他们知道自己并非一无是处（not wrong all of the time）。如果对其工作吹毛求疵，将会降低他们对你的尊重。可以提出建设性的批评建议，但不要强求每个解决方案看起来好像都出自你手。

以正确的方式经营开发团队，其重要性不言而喻。团队成员们并肩作战共同负重。要对他们一视同仁，公平对待。应当确保团队具有打不垮的首发阵容（starting lineup），而一旦已经拥有“常胜铁军”，就要竭力维持。

作者简介见第 111 页。

软件并非真实的存在

Software Doesn't Really Exist

查德·拉·瓦因（Chad La Vigne）



软件工程经常被拿去和完善的传统学科——如土木工程——进行对比。这些类比有个问题，和这些传统实践制造的有形产品不同，软件并非真实的存在。总之，不是传统意义上的那种存在。我们在 1 和 0 的世界中，并不受与在传统工程范式（classic engineering paradigms）中相同的物理规则的约束。虽然在软件设计阶段可以应用工程原理，但是，假设能套用传统工程的方法来实现设计，则是不现实的。

业务和软件都是活生生、会变化的实体。业务需求可能会因新近收购的业务伙伴和营销战略而迅速变化。在软件项目中，很难使用与传统工程——如桥梁建造——中使用的相同方法。在桥梁建造项目中，不太可能会在项目进行到一半时，要求把桥梁的位置移动到别处。但是，收购活动很可能会要求应用增加对基于组织结构（organization-based）的内容管理的支持。比较时，要把这些方面也考虑进去。我们常说软件架构的决策很难改变，但并非像一成不变地嵌入石头之中的那些东西一样。

应该领悟到我们构建的产品是柔韧的（pliable），并且围绕产品的需求很可能会发生变化，可以让我们站在另外的位置上看待构建活动，这个位置不同于那些构建固定物体（immoveable object）的人所站的。物理实体性质的工程活动，按照“对工作进行计划，按计划开展工作（plan the work, work the plan）”的自然方式进行实施，是相对容易的。但对于软件而言，更多的需要以“对工作进行计划，不断调整计划（plan the work, massage the plan）”的方式处理。

这些差异并非总是坏事，有时它们也是利好消息。例如，不必非得受限于以特定顺序来构建软件系统的组件，如此便可首先解决高风险的问题。这和桥梁建造这样的事情形成鲜明对比，它们在物理上的许多限制，都是由于任务需要按一定次序完成才行。

然而，软件工程的灵活性也表现出一些问题，其中许多是自寻烦恼。作为架构师，我们对自己的手艺品（craft）具有“软（soft）”的天生特性心知肚明，而且我们喜欢解决问题。但糟糕的是，业务方也模糊地知道这些事实。因此他们极易推动大的变化。不要仅因为它召唤起你热衷提供解决方案的天性（solution-providing nature），就急于接受吸纳巨大的架构性变更。这样的决定会破坏原本健康的项目。

记住，需求文档不是蓝图（译注 1）（blueprint），软件并非真实的存在。我们创造的虚拟物，相较其物理世界中的对应物（counterpart），更易于改变。这是一件好事，因为很多时候对它们确有这样的需求。以我们仿佛正在构建不可移动物品的方式来做计划，这当然没有问题；只是，受命移动那些不可移动物品时，我们不能感到万分诧异或毫无准备。

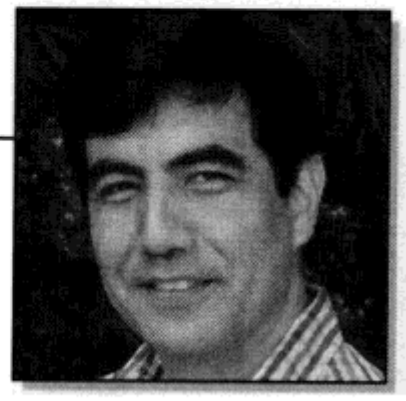
作者简介见第 111 页。

译注 1：建筑设计师的施工图、机械设计工程师的加工图等传统工程领域的图纸，要晒成蓝色的图纸，所以称为蓝图。

学习新语言

Learn a New Language

伯克哈特·赫夫纳盖尔（Burkhardt Hufnagel）



作为成功的架构师，必须要让别人能够理解你，他们可能不懂你的“母语”。不，我不是要建议你去学习世界语（Esperanto）或克林贡语（Klingon）（译注 1），但对于业务和测试方面的基本用语(basic Business, and Testing)，你至少应该会。如果对于程序员用语（Programmer）还不流利，那应将之排在第一优先级。

如果还没有看出学习其他“语言”的价值，设想以下情况。业务人员希望对现有系统做些改变，因此，他们召集架构师和程序员开会对此进行讨论。不幸的是，技术团队中没人会说业务的行话，业务人员中也没人会说程序员的行话。这次会议很可能会是如下情形：

- 业务人员讲了一分钟，提出一个需求，期望能对某现存产品做一个相对简单的改进，并解释为何这个改变将有助于销售团队增加市场份额和影响力。
- 业务人员还在讲的时候，架构师开始在草纸上画起神秘的符号，和一名程序员开始以令人捉摸不透的方式在纸面上进行无声无息的争论。
- 业务人员终于说完了，满脸期待看着架构师。
- 低声争论完后，架构师走到白板边，开始一边画一些代表现存系统多个视图的复杂图形，一边（以复杂的技术术语）解释，为何在没有对系统做出重大变化之前，预计无法实现如其所请的改进，想要做到，则可能要重新设计和重写整个系统。

译注 1：克林贡语（Klingon Language）是除了世界语之外最完善的人造语言。这套语言是为了 20 世纪末期美国著名的科幻电影和连续剧《星际迷航》(Star Trek)而发明的。在影片中，使用这种语言的克林贡人是一个掌握着高科技却野蛮好战的外星种族。克林贡语的发明者是美国语言学家马克.欧克朗（Marc Okrand)。语法来自北美印第安人部落,-接近于阿滋特克语系。它采用了自然语言中最少见 OVS 语序，即宾语-谓语-主语。这与绝大部分自然语言的语序相反。克林贡语的发明者希望通过这种少见的语序来获得外星种族语言的神秘感。克林贡语使用标准的 26 个拉丁字母，大部分单词长度比较短。

- 业务人员（对这些图形很难理解，这些解释也听不懂多少）公开表示震惊和难以置信，如此简单的事情居然要做出如此巨大的变化。他们开始怀疑架构师是否是认真的，或者只是无中生有来试图躲避改动吧。
- 同时，架构师和程序员也同样感到诧异，业务人员居然没有看到，要满足所谓的“小”调整将需要对核心系统的功能做出重大的修改。

问题就在这里。没有一方能够理解另一方的意思，也不理解对方言论里一半的词语。不信任和误解由此产生。相比和自己不同的人在一起，人们与那些和自己类似的人相处感觉会更舒服，这是基本的心理学规律。

想象一下，如果架构师能够以业务人员可理解的术语向业务人员解释其中的问题，以程序员可理解的术语向程序员转述业务上的问题，上述情况可能会大有改观。其结果可能就不是诧异和不信任，而是同意和允准。

这并不是说，掌握多种“语言”就可以解决所有的问题，但定将有助于防止沟通不畅和误解，避免导致问题。

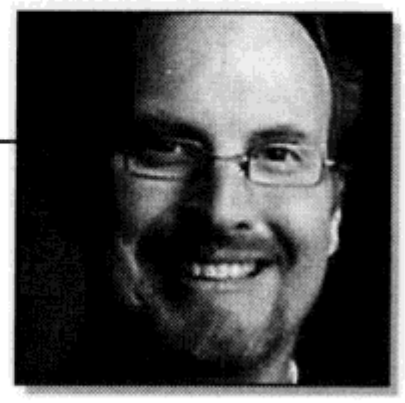
要对变化是否有意义拍板的架构师们，祝你们成功！或者，用克林贡语说，Qapla！

作者简介见第 175 页。

没有永不过时的解决方案

You Can't Future-Proof Solutions

理查德·蒙森-哈费尔（Richard Monson-Haefel）



今天的解决方案会成为明天的问题

Today's Solution Is Tomorrow's Problem

没有人能够预知未来。如果你认为这是普遍真理，那么问题就变成，究竟前面多远算是未来？10 年？两年？20 分钟？如果无法预知未来，那么也无法预知此刻以后的任何东西。所能知道的，只有当下及这一时刻之前的事情。要知道下一刻，只有等它到来。这也是发生车祸的原因，如果已经知道周四会发生意外，人们大概就会留在家中不出去了。

然而，我们看到很多软件架构师试图设计出“永不过时（future-proof)”（恕我找不到更好的词语描述）的系统。根本不可能存在“永不过时”的架构。不管现在做出什么样的架构决策，这种选择最终会过时。现在用的很酷的编程语言，最终将变成明天的 COBOL。今天的分布式框架，将会变成明天的 DCOM。总之，今天的解决方案一定会变成明天的问题。

今天做出的选择，在未来很大程度上会是错误的。如果接受这一事实，便可放下试图获得永不过时的架构的负担了。如果今天做出的任何选择，在未来会是糟糕的选择，那就不要操心将来要怎样的东西——只要选择能满足当前需求的最佳解决方案就行了。

“分析瘫痪（analysis paralysis）”是今天架构师们碰到的问题之一，此问题最大的归因，是试图去猜测对未来而言最好的技术。为当前选择一项好技术已属困难，要选择在未来也切合可用的好技术只会徒劳无功。仔细查看目前业务所需为何，以及当前技术市场提供的东西。从中选择能够满足当前需求的最好解决方案，因为别的东西，不仅对明天是错误的选择，而且，对今天就已是错误的选择。

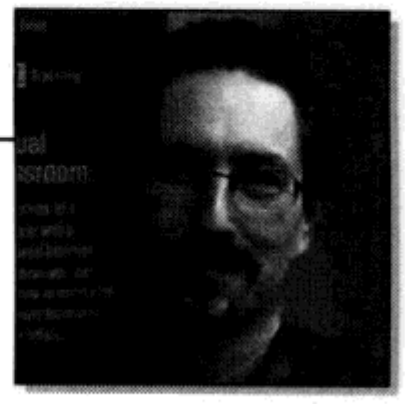
作者简介：

理查德·蒙森-哈费尔（Richard Monson-Haefel）是一名独立软件开发者，与人合著过《Enterprise JavaBeans》的全部5个版本，以及《Java消息服务》的两个版本（均由O'Reilly出版）。他是一名多点触摸界面的设计师/开发者，在企业计算领域是领先的专家。

用户接受度问题

The User Acceptance Problem

诺曼·卡诺瓦利（Norman Carnovale）



人们并不总是满心欢喜地接受新系统或系统的重大升级。这种情势，可能会对项目的成功构成威胁。

人们不同意实施新系统的决定，这并不罕见，尤其在初期时。这在预期之中，也情有可原。但是，相比持续地消极应对，人们对新系统的最初反应通常是漠不关心。

架构师的目标，是要去了解和衡量接受度问题（acceptance problem）带来的威胁，并朝能减轻这些威胁的方向开展工作。要做到这一点，必须认识到这些问题，并深思其中的原因。较常见的原因有：

- 人们可能会关注需要新系统（和随之而来的旧系统的退出）的必要性问题。还包括对角色转变时可能会失去职能、影响力或权力的恐惧。
- 对新的（未经证实的）技术怀有恐惧。
- 成本/预算方面的考虑。
- 人们根本就不喜欢变革。

要针对每种原因，采用不同的可行办法，其中有一些你可以解决，而另外一些你则无能为力。必须认识到其中的差异，并快速处理掉那些你能够解决的问题。尽早开始和最终用户讨论新系统，包括其真正的和潜在的好处与不足。最有效的长期解决办法，是运用系统设计本身来消解个中忧虑。其他有效的解决方法，包括培训、定期安排系统的演示（在项目生命周期的早期阶段就开始），并分享用户从新系统中将会获得的知识。

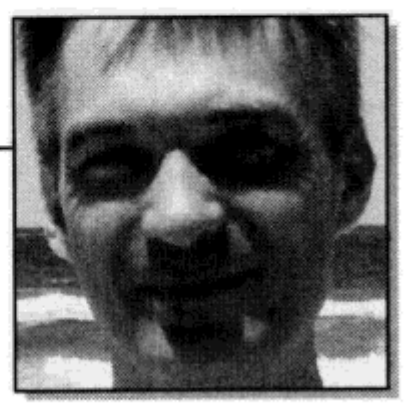
“项目拥戴者”可以帮助消除用户接受度问题。这个职位最理想的人选，是能代表用户或利益相关方的人，有时候，他要说服自己“就是”用户或利益相关方。如果没有这样的人，那么在一开始就要努力寻找这样的人。一旦召集到项目拥戴者，就要尽力给予支持。

作者简介见第 43 页。

清汤的重要启示

The Importance of Consommé

埃本·休伊特（Eben Hewitt）



清汤（consommé）（译注 1）是一种非常清澈的肉汤，通常以牛肉或小牛肉烹制而成，极其鲜美精妙。上品清汤非常澄澈。烹制清汤公认颇具挑战性，而且耗时长久，因为要去除会弄浑清汤的脂肪和其他固态物，获得绝对澄澈之上品，只有一个方法：依靠简单的、不断重复地精炼浓缩。一遍又一遍地浓缩，精益求精，最终烹出上品美味。品尝清汤，就仿佛在吸啜精华。事实上，这正是此道菜肴的精华。

在美国的烹饪学校，对学员厨师的一个简单测试便是通过烹制清汤进行鉴定：老师在学员烹制的琥珀肉汤中丢下一个 10 美分的硬币，如果沉在碗底的硬币上的日期清晰可见，考试通过；如果不能，便告失败。

设计软件架构也应当不断精炼浓缩，反思各种构想，直至可以确定系统中每个需求的本质。要像拿着约里克（Yorick）头骨的哈姆雷特（Hamlet）一样（译注 2），问：这是什么？它的属性是什么？它们之间的关系又如何？澄清概念，使架构内的各种关系确凿无误，内在一致。

软件中许多遗漏的需求和缺陷，可最终追溯到含糊笼统的语言描述上。向客户、开发人员、分析师和用户，不断重复地询问同样的问题，直到他们不堪其扰昏昏欲睡。然后，换不同的方式对问题改头换面，就像在“托词（alibi）”中寻找蛛丝马迹的律师一样，梳理出所有的新发现、分歧或矛盾。不断精炼。

译注 1：consommé，是法国菜式中的清汤，法国清汤固定以 12 小时小火熬煮加上多次过滤烹调而成，大量营养成分及胶原蛋白皆溶于清澈无油浓郁鲜美的汤汁中，人体非常容易吸收。

译注 2：莎士比亚戏剧中出现骷髅头的最著名场景在《哈姆雷特》这部戏剧中。哈姆雷特王子在一个墓地中和掘墓人交谈，同时发现挖出来的一个骷髅头是以前熟识的宫廷小丑尤里克（Yorick）的头颅，哈姆雷特于是手捧这个宫廷小丑的骷髅头，发出了一番人生无常的独白感慨。

要关注哪些概念可以从架构中移除，哪些名词构成了这些概念，确定它们的本质。要以外科手术般的精确态度对待需求描述，拒绝那些模棱两可、笼统、毫无根据的假设或无关的废话。这将有助于使概念更加丰富，更加健壮稳定。浓缩才是精华。

可以借助询问："如果我加上'总是，永远，不管任何情况下'，你对此还会做出同样的陈述（statement）吗？"对陈述进行测试。对于如此绝对的承诺，人们通常会有所抵制，因此必然会进一步完善措辞。通过这种语言上的筛子（a linguistic sieve），迫使其中的概念显现出来，并澄清概念。不断重复，直至只剩下可信陈述的详尽列表，这些可信陈述（true statement）简单、可核实，描述了系统的本质特性。

如果你能透过清汤，看到锅底硬币上的日期，你就知道架构已臻完成。

作者简介见第 161 页。

对最终用户而言，界面就是系统

For the End User, the Interface Is the System

维纳亚克·赫格德（Vinayak Hegde）



糟糕的用户界面埋没了太多的好产品。最终用户（the end user）通过用户界面访问系统。如果产品的用户交互体验质量让人无法忍受，那么，无论产品在技术上如何先进或具有开创性（path-breaking），都会造成让人无法忍受的印象。

用户界面是架构的重要组成部分，但往往也是被忽视的部分。架构师应借助于专家的服务，诸如用户体验设计师和可用性专家。用户交互专家和架构师一起工作，则可驱使产品的界面设计和内部机制的配合天衣无缝。在早期阶段就让用户界面专家参与其中并贯穿整个产品开发阶段，能确保用户界面和产品浑然一体，使最终产品整洁优美。当产品还在 beta 阶段时，架构师应仔细观察由真实的最终用户完成的用户交互测试，并将他们的反馈纳入最终产品中。

随着时间变迁，技术会发生变革，产品特性也会不断增加，产品的用法也会因此经常发生变化。架构师应确保，随着架构变化而变的用户界面也要能够反映用户的期望。

用户交互应是整个产品架构的目标之一。事实上，用户交互应和健壮性（robustness）和性能（performance）一样，是架构权衡（tradeoff）和内部产品文档化（documentation）这一决策过程的组成部分。随着时间而变的用户交互设计，也应当像代码一样被记录下来。尤其值得一提的是，如果用户界面和产品其余部分采用不同的编程语言，更应如此。

不仅要让最常用的交互易用，而且要使最终用户能够从中获得回报，这是架构师的责任。好的用户界面能够帮助客户提高生产力，客户因此会更加开心。如果产品能够帮助人们变得更加高效，也将有利于产品自身的业务收益。

作者简介：

维纳亚克·赫格德（Vinayak Hegde）是一名奇客（geek），他的兴趣是计算机、摄影和创业。目前他是 Akamai Technologies 公司的一名架构师。

优秀软件不是构建出来的，而是培育起来的

Great Software Is Not Built, It Is Grown

比尔·德·霍拉（Bill de hÓra）



作为架构师，负有初始化软件系统的结构并对之进行规划的责任。软件系统会随着时间生长变化，要返工修改，要能和其他系统通信交互——而这一切几乎都会以你和利益相关者无法预见的方式发生。虽然我们顶着架构师的头衔，也从建筑和工程领域里借用了许多隐喻（metaphor），但是优秀软件不是构建出来的，而是培育起来的。

单凭规模大小，便可窥见软件成败的端倪；细思之下便可知，在一开始就要设计庞大的系统几乎毫无好处可言。然而，有时我们都会无法抵制如此行事的引诱。除了易于招致额外的复杂性和惯性，前期便要设计庞大的系统，意味着项目会变得更为大型，而大型项目更可能会失败，更可能无法验证，更可能脆弱不堪（fragile），更易产生不必要和无用的产物，成本可能更是无比高昂，而且，更易招致不利的政治因素。

因此，无论多么诱人，都要抵制住试图设计出庞大完整的系统，来“满足甚或超出”已知需求和特性期望的想法。要有宏伟的远景（grand vision），但不要有庞大的设计。环境和需求的变化不可避免，你和你的系统都要学会适应变化。

如何做到这一点？确保软件系统能够演化和具备适应性的最好办法，是从一开始就做到这样。引导系统进行演化，意味着要从一个小型但可用的系统（a small running system）开始，这个系统是预期架构一个可用的子集，亦即最简单的可行实现。这一新生系统拥有很多期望的特性，在架构上可以教给我们很多东

西，这些是庞大系统和一大堆架构文档永远无法给予我们的。很可能你自身也已参与到系统的实现中了。由于没有庞大的体表面积，这样的系统更易测试，耦合性也会更低。因此，开发团队也会相对较小，项目协调所需的开销由此也可降低。其特性也更为清晰易见。系统更易于部署。你和团队也会在第一时间知道什么可行什么不可行。它会揭示出，系统在哪里已经僵硬固化、变得脆弱甚或可能断裂，因而无法轻松进行演化。也许，最为重要的是，由于利益相关者从一开始便可参与到整体设计中，系统对他们会显得实在确凿，易于理解。

设计尽可能小的系统，帮助成功交付，并推动它向宏伟的远景目标不断演化。虽然听起来似乎是放弃了控制权，甚至是在逃避责任，但是最终，利益相关者会感谢你。注意，不要把这种演化式方法和削减需求、规范混乱和生产废品这样的做法混淆在一起。

作者简介见第 117 页。

索引
Index

Numbers

1,000-foot view, 56–57

A

accidental complexity, 4–5
ambiguity, 190
analogies, 118–119
Analysis Paralysis, 85
Anderson, Dave, 130–131
anti-patterns, 85
application architecture patterns, 84
application support and maintenance, 114–115
architects
 as developers, 126–127, 150–151
 as kings, 88–89
 characteristics of good, 38
architectural tradeoffs, 44–45
architecture and design patterns, 84–85
Architecture Tradeoff Analysis Method (ATAM), 45
assumptions, 106–107
axioms, 118–119

B

Bartlett, David
 Continuously Integrate, 40–41
 Janus the Architect, 98–99
binary decisions versus real world, 94–95
Booch, Grady, 48
Borwankar, Nitin, 2–3
boundaries, 100–101
Braithwaite, Keith
 Build Systems to Be Zuhanden, 178
 Learn from Architects of Buildings, 90–91
 Quantify, 20–21
 There Can Be More Than One, 32
Brewer's conjecture, 116
Brooks Jr., Fredric, 62
Brown, Mike
 Before Anything, an Architect Is a Developer, 126–127
 If You Design It, You Should Be Able to Code It, 150–151
budgetecture, 18–19
build early and often, 40
building architects, learning from, 90–91
business case, 170–171
business domains, understanding, 60–61
business enterprise application development, 34–35

C

call-stack architecture, 95
Carnovale, Norman
 Avoid Scheduling Failures, 42–43
 The User Acceptance Problem, 188–189
Chak, Dan, 46–47
challenges, 66
change, impact of, 134–135
clarity, 8–9

cleverness, 160–161
code versus design, 22
code versus specifications, 22
coding, 150–151
Cofsky, Evan, 88–89
commit-and-run, 30–31
communication, 6–7
 effective, 8–9
 reuse, 52–53
 standing up, 14–15
complexity, 124–125
 accidental, 4–5
 essential, 4–5
 reducing, 134–135
components
 creating well-defined interfaces between, 76
 deploying, 76
 "early release" versus "incremental deployment", 77
Consistency, Availability, and Partitioning (CAP), 116
consommé, 190–191
constraints, 116–117
content, great, 144–145
context, 86–87
contextual sense, 24–25
continual refinement of thought, 190–191
continuous integration, 40–41, 172
control freak architect, 96
Cost Benefit Analysis Method (CBAM), 45
cost-cutting, 18–19
Crawford, Doug, 134–135
criteria, 20–21
criticism, 54–55
customers
 balancing interests with technical requirements, 28–29
 interfaces, 192–193
 needs, 2–3
 requirements, 164–165
 seeking value in requirements, 12–13
 user acceptance, 188–189

D

Dahan, Udi, 14–15
database, as fortress, 46–47
data, controlling, 172–173
data models, 46
data-oriented perspective, 122–123
Davies, John, 38–39
death march, 85
defining your work, 152–153
de hÓra, Bill
 Great Software Is Not Built, It Is Grown, 194–195
 Prepare to Pick Two, 116–117
design
 large complete systems, 194–195
 uncertainty, using as driver, 48–49
 versus code, 22
design by committee, 85
design patterns, 84–85, 110–111
design process, 166–167
developers
 architects as, 126–127, 150–151
 autonomy, 64–65
 empowering, 102–103
diagrams, 8
diligence, 156–157
documentation, 104–105
Doernenburg, Erik, 56–59
domain-driven design, 25
duplication, 92–93

E

egos, 54–55
empowering developers, 102–103
enterprise architecture patterns, 84
essential complexity, 4–5
ethical consequences, 74–75
Evans, Eric, 100
event-driven architecture (EDA), 84

F

fallibility, 16–17
Ford, Neal, 4–5
Fowler, Martin, 40, 85
frameworks
 compatibility, 168–169
 reducing complexity, 5
 trying before choosing, 58–59
future-proofing solutions, 186–187

G

Gardiner, Sam
 A Rose By Any Other Name Will End Up As a Cabbage, 152–153
 Stable Problems Get High-Quality Solutions, 154–155

Garson, Edward
Context Is King, 86–87
Heterogeneity Wins, 78–79
Gawande, Atul, 157
generality after simplicity, 36–37
Gillard-Moss, Peter, 166–167
good enough versus perfect, 140–141
good ideas, dangers of, 142–143
GraphViz, 57
great content, 144–145

H

hardware considerations, 136–137
Harmer, Michael, 118–119
Hart, Brian, 156–157
Hawkins, Barry
The Title of Software Architect Has Only Lowercase a's, 68–69
Value Stewardship Over Showmanship, 72–73
Hawthorne, Eric, 168–169
Hegde, Vinayak, 192–193
Heidegger, Martin, 178
Henney, Kevlin
Simplicity Before Generality, Use Before Reuse, 36–37
Use Uncertainty As a Driver, 48–49
heterogeneity, 78–79
heuristic approach, 24
Hewitt, Eben
Don't Be a Problem Solver, 176–177
Don't Be Clever, 160–161
The Importance of Consommé, 190–191
Your Customer Is Not Your Customer, 164–165
High, Timothy
Challenge Assumptions—Especially Your Own, 106–107
Empower Developers, 102–103
If There Is Only One Solution, Get a Second Opinion, 132–133
Record Your Rationale, 104–105
Hillaker, Harry, 12
Hohpe, Gregor
Don't Control, but Observe, 96–97
Welcome to the Real World, 94–95
Homer, Paul W.
It Is All About The Data, 122–123
Share Your Knowledge and Experiences, 108–109
Hufnagel, Burkhardt
Learn a New Language, 184–185
Pay Down Your Technical Debt, 174–175

I

impact of change, 134–135
industry trends, 60
InfoViz toolkit, 57
Ing, David, 112–113
ingenuity, 156
integration patterns, 84, 85
interfaces, 100–101, 192–193
interoperability, 79

J

Janus (Roman god), 98–99
Jones, Stephen, 148–149

K

Kasper, Mncedisi, 114–115
key dimensions, stretching, 148–149
kings, architects as, 88–89
knowledge and experiences, sharing, 108–109
Koenig, Andrew, 85

L

Landre, Einar
Architects' Focus Is on the Boundaries and Interfaces, 100–101
Programming Is an Act of Design, 62–63
Seek the Value in Requested Capabilities, 12–13
languages, learning multiple, 184–185
LaVigne, Chad
Choose Your Weapons Carefully, Relinquish Them Reluctantly, 162–163
Control the Data, Not Just the Code, 172–173
Find and Retain Passionate Problem Solvers, 180–181
Make Sure the Simple Stuff Is Simple, 124–125
Pattern Pathology, 110–111
Software Doesn't Really Exist, 182–183
The Business Versus the Angry Architect, 146–147

leadership, 8–9
legacy, designing for, 130–131
libraries, trying before choosing, 58–59
Lovelace, Richard, 165

M

maintenance, application, 114–115
Malamidis, George, 128–129
managing risks, 51
Mcphee, Scot, 138–139
metaphors, 112–113
metrics, 171
Meyer, Jeremy, 52–53
Monson-Haefel, Richard, xv–xviii, 186–187
Muirhead, Dave, 34–35
multiple solutions, 33–34, 132–133
mushroom management, 85

N

names conveying intentions, 152–153
negotiation, 18–19
Nelson, Philip
 Give Developers Autonomy, 64–65
 Time Changes Everything, 66–67
Nilsson, Niclas
 Commit-and-Run Is a Crime, 30–31
 Fight Repetition, 92–93
Nyberg, Greg
 Avoid "Good Ideas", 142–143
 "Perfect" is the Enemy of "Good Enough", 140–141
Nygaard, Kristen, 62
Nygard, Michael
 Engineer in the White Spaces, 82–83
 Everything Will Ultimately Fail, 16–17
 Skyscrapers Aren't Scalable, 76–77
 Software Architecture Has Ethical Consequences, 74–75
 You're Negotiating More Often Than You Think, 18–19

O

objective criteria, 20–21
observation, 96–97
one-size-fits-all solutions, 24–25
overconfidence, 146–147
overengineered frameworks, 4–5

P

Parsons, Rebecca, 26–27
patterns, 25
 design, 84–85, 110–111
perfect versus good enough, 140–141
performance, 10–11, 80–81
performance testing, 26–27
pipeline architecture, 84
polyglot programming, 78–79
Poppendieck, Mary, 58
Poppendieck, Tom, 58
principles, 118–119
prioritizing, 71
problems and strategies, 50–51
problem solvers, 180–181
problem-solving mode, 176–177
productivity, 80–81
professionalizing the practice, 68–69
programming, design versus construction, 62–63
project scope, 70–71

Q

quantitative criteria, 20–21
Quick, Dave
 Scope Is the Enemy of Success, 70–71
 There Is No 'I' in Architecture, 54–55
 Warning: Problems in Mirror May Be Larger Than They Appear, 50–51

R

Ramm, Mark, 6–7
Randal, Allison, 22–23
real world, 94–95
repetition, 92–93
requirements, 2–3, 70, 164–165
 balancing interests with technical requirements, 28–29
 seeking value in, 12–13
resource-oriented architecture (ROA), 84
responsibility, 158–159
return on investment (ROI), 34, 128–129
reuse, 52–53
Richards, Mark
 Architectural Tradeoffs, 44–45
 Communication Is King, 8–9
 Talk the Talk, 84
 Understand the Business Domain, 60–61

risks, managing, 51
Russell, Craig, 80–81

S

scheduling, avoiding failures, 42–43
scope, 70–71
service-oriented architecture (SOA), 84
Shank, Clint, 120–121
sharing knowledge and experiences, 108–109
shortcuts, 138–139
showmanship versus stewardship, 72–73
simplicity, 66–67
 before generality, 36–37
 context, 86
 perfect versus good enough, 140–141
simplifying, 124–125
 reducing complexity, 4–5, 134–135
social interactions, 6–7
software, 182–183
 failure, 194
Software Engineering Institute (SEI)
 websites, 45
software patterns, 25
solutions
 future-proofing, 186–187
 multiple, 33–34, 132–133
 one-size-fits-all, 24–25
 problem-solving mode, 176–177
 stabilizing problems, 154–155
source code control, 172
specifications versus code, 22
stabilizing problems, 154–155
Stafford, Randy
 Application Architecture Determines Application Performance, 10
 Architecting Is About Balancing, 28–29
 There Is No One-Size-Fits-All Solution, 24–25
stakeholders, balancing interests with technical requirements, 28–29
 (see also customers)
standing up, 14–15
stewardship versus showmanship, 72
stretching key dimensions, 148–149
support, application, 114–115
system response time, 81

T

technical debt, 174–175
technical testing, 27
technologies, selecting, 162–163
testing
 performance, 26–27
 technical, 27
tools
 trying before choosing, 58–59
 zuhanden, 178–179
tradeoffs, architectural, 44–45

U

uncertainty, using as driver, 48–49
user acceptance, 188–189

V

value
 proposition, 170
 seeking value in requirements, 12–13
Vasa, 44–45
Visio, 8

W

Wadia, Zubin, 144–145
walking skeleton, 120–121
Waterhouse, Randy, 88
Wethern's Law of Suspended Judgment, 106
whiteboard meetings, 8
white spaces, engineering, 82–83
Wickramanayake, Kamal, 136–137

Z

Zhou, Yi
 Make a Strong Business Case, 170–171
 Take Responsibility for Your Decisions, 158–159
zuhanden, 178–179